

**USENIX Association**

**Proceedings of the  
17th USENIX Symposium on Operating Systems  
Design and Implementation (OSDI '23)**

**July 10–12, 2023  
Boston, MA, USA**

© 2023 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-34-2



## Symposium Organizers

### Program Co-Chairs

Roxana Geambasu, *Columbia University*  
Ed Nightingale, *Apple*

### Program Committee

Atul Adya, *Databricks*  
Rachit Agarwal, *Cornell University*  
Nitin Agrawal, *Google*  
Ramnatthan Alagappan, *University of Illinois at Urbana–Champaign*  
Jeremy Andrus, *Apple*  
Sebastian Angel, *University of Pennsylvania*  
Mahesh Balakrishnan, *Confluent*  
Adam Belay, *MIT CSAIL*  
Emery Berger, *University of Massachusetts Amherst*  
Edouard Bugnion, *EPFL*  
George Candea, *EPFL*  
Kang Chen, *Tsinghua University*  
Vijay Chidambaram, *The University of Texas at Austin and VMware Research*  
Mosharaf Chowdhury, *University of Michigan*  
Byung-Gon Chun, *Seoul National University and FriendliAI*  
Asaf Cidon, *Columbia University*  
Manuel Costa, *Microsoft Research*  
Landon Cox, *Microsoft Research*  
Natacha Crooks, *University of California, Berkeley*  
Jon Crowcroft, *University of Cambridge*  
Heming Cui, *University of Hong Kong*  
Dilma Da Silva, *Texas A&M University*  
Murat Demirbas, *Amazon Web Services*  
Ittay Eyal, *Technion—Israel Institute of Technology*  
Jason Flinn, *Meta*  
Bryan Ford, *EPFL*  
Aishwarya Ganesan, *University of Illinois at Urbana-Champaign and VMware Research*  
Phillip Gibbons, *Carnegie Mellon University*  
Yossi Gilad, *Hebrew University of Jerusalem*  
Joseph Gonzalez, *University of California, Berkeley*  
Andreas Haeberlen, *University of Pennsylvania and Roblox*  
Steven Hand, *Google*  
Tim Harris, *Microsoft*  
Wenjun Hu, *Yale University*  
Ryan Huang, *Johns Hopkins University*  
Rüdiger Kapitza, *Friedrich-Alexander-Universität Erlangen-Nürnberg*  
Brad Karp, *University College London*  
Baris Kasikci, *University of Michigan*  
Eddie Kohler, *Harvard University*  
Mathias Lécuycer, *University of British Columbia*  
Philip Levis, *Google and Stanford University*  
Amit Levy, *Princeton University*  
Jinyang Li, *New York University*  
Hyeontaek Lim, *Google*  
Wyatt Lloyd, *Princeton University*  
Jay Lorch, *Microsoft Research*  
Shan Lu, *University of Chicago*  
Martin Maas, *Google*  
Jonathan Mace, *Max Planck Institute for Software Systems (MPI-SWS)*  
Ratul Mahajan, *University of Washington and Intentionet*  
Z. Morley Mao, *University of Michigan and Google*

James Mickens, *Harvard University*  
Thomas Moscibroda, *Microsoft*  
Deepak Narayanan, *Microsoft Research*  
Ravi Netravali, *Princeton University*  
Jason Nieh, *Columbia University*  
Cristina Nita-Rotaru, *Northeastern University*  
Shadi Noghahi, *Microsoft Research*  
Aurojit Panda, *New York University*  
Kyoungsoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*  
Bryan Parno, *Carnegie Mellon University*  
Daniel Peek, *Meta*  
Peter Pietzuch, *Imperial College London*  
Dan Ports, *Microsoft Research*  
Costin Raiciu, *University Politehnica of Bucharest*  
David Richardson, *Apple*  
Luis Rodrigues, *INESC-ID and Instituto Superior Técnico, University of Lisbon*  
Christopher Rossbach, *The University of Texas at Austin and Katana Graph*  
Malte Schwarzkopf, *Brown University*  
Marco Serafini, *University of Massachusetts Amherst*  
Marc Shapiro, *Sorbonne-Université–LIP6 and Inria*  
Ji-Yong Shin, *Northeastern University*  
Mark Silberstein, *Technion—Israel Institute of Technology*  
Alex C. Snoeren, *University of California, San Diego, and Google*  
Ion Stoica, *University of California, Berkeley*  
Ryan Stutsman, *University of Utah*  
Steven Swanson, *University of California, San Diego*  
Adriana Szekeres, *VMware Research*  
Kaushik Veeraraghavan, *Facebook*  
Geoffrey M. Voelker, *University of California, San Diego*  
Roger Wattenhofer, *ETH Zurich*  
Michael Wei, *VMware Research*  
Yubin Xia, *Shanghai Jiao Tong University*  
Tianyin Xu, *University of Illinois at Urbana–Champaign*  
Junfeng Yang, *Columbia University*  
Ding Yuan, *University of Toronto and YScope*  
Lidong Zhou, *Microsoft Research*

### Poster Session Co-Chairs

Ryan Huang, *University of Michigan*  
Adriana Szekeres, *VMware Research*

### Steering Committee

Marcos K. Aguilera, *VMware Research*  
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*  
Angela Demke Brown, *University of Toronto*  
Jason Flinn, *Meta*  
Casey Henderson, *USENIX Association*  
Jon Howell, *VMware Research*  
Kimberly Keeton, *Google*  
Hank Levy, *University of Washington*  
Jay Lorch, *Microsoft Research*  
Shan Lu, *University of Chicago*  
James Mickens, *Harvard University*  
Timothy Roscoe, *ETH Zurich*  
Margo Seltzer, *University of British Columbia*  
Geoff Voelker, *University of California, San Diego*  
Hakim Weatherspoon, *Cornell University and Exotanium, Inc.*

## External Reviewers

Hans-J. Boehm

Michael Hicks

Jon Howell

David Mazières

Rohan Padhye

## Message from the OSDI '23 Program Co-Chairs

Dear Colleagues,

Welcome to the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23).

Once again, we are co-locating OSDI and the USENIX Annual technical Conference (USENIX ATC '23). We are excited to be back in person in these two forums for such a large gathering of technical experts across academia and industry. We hope you come away from OSDI and USENIX ATC inspired, refreshed and re-engaged with the wider systems research community!

This year, OSDI received 255 submissions. We accepted 50 submissions, which is a 19.6% acceptance rate. This is inline with the 253 submissions and 19.4% acceptance rate from last year. In addition to the 50 papers accepted this year, 5 additional papers were accepted from the OSDI '22 Revise and Resubmit process. This brings the total program to 55 papers presented at OSDI across an incredibly wide breadth of areas of focus. Although this is 6 more papers than last year, we continue to be committed to a single-track conference at OSDI and have put together a 3-day program for all 55 papers.

Given the large number of submissions, we formed a program committee of 86 members in addition to the two program co-chairs. There were also 5 external reviews where a particular area of expertise or evaluation was needed on a single paper. We are very grateful to the program committee for their diligence, focus, and optimism during the review process. OSDI is known for high quality reviews that help and inform the authors to produce their very best work.

The program committee reviewed submissions in two rounds. Every paper received at least three reviews in the first round. Roughly 40% of the papers were rejected during the first round of reviews. One paper was accepted after the first round of reviews concluded, and the remainder advanced to the second round. Papers then received 2 or 3 additional reviews. Subsequently, the PC had an asynchronous online discussion phase where an additional 35 papers were accepted and 77 papers were rejected. Thirty six papers were advanced to a synchronous, online two-day PC meeting. We purposefully organized the PC meeting by topic area, which allowed experts who might not have a paper in a particular area up for discussion, to dial in and participate in the areas most relevant to them. We finished the PC meeting on time, and had a great discussion from PC members all over the world. Each accepted paper was assigned a shepherd to work with the authors to revise the paper in response to reviewer feedback. The committee completed more than 1,000 reviews and posted hundreds of comments as part of the online discussion process. Finally, we did allow one of the program co-chairs to submit a paper. However, the paper was assigned a "paper administrator," and neither co-chair had any involvement in the review of that paper, and both were informed of the paper's outcome after the PC meeting concluded.

The program committee also chose to continue the revise-and-resubmit process. A small set of papers were recommended by the program committee to undergo this process. Those authors will have a much longer period of time (until August) to revise their papers, add important missing details, and/or run additional experiments, as requested by the PC. The original reviewers will review the revised submission. If accepted, the authors will be allowed to submit a "camera-ready" version of the paper in October of 2023 and then the authors will present the paper and have that paper officially included in the OSDI 2024 program. This early camera ready deadline allows authors to discuss the work (and have others cite it) well before the OSDI 2024 program appears.

Once the program was completed and the camera-ready deadline had passed, we began the process of selecting the Jay Lepreau Best Paper Award. We began by asking PC members for nominations and looking at the top-rated papers during the review process. We formed a small group of non-conflicted PC members who read each paper and agreed on the best paper of those chosen for consideration.

OSDI '23 had an artifact-evaluation committee that organized and evaluated the artifacts submitted by authors. The committee co-chairs this year are Jianyu Jiang, Nathan Rutherford, and Cesar A. Stuardo. Thirty-two papers submitted artifacts supporting their research. Twenty-nine papers received the Available badge. Thirty-one papers received the Functional badge, and 27 papers received the Reproduced badge, meaning the results in the paper were independently reproduced by the committee.

OSDI '23 had a poster submission process organized by co-chairs Adriana Szekeres and Ryan Huang. Thirty-three posters were accepted for display at the OSDI poster session.

Finally, as PC chairs, we rely on so many dedicated volunteers and professional staff to make this conference a reality. We thank the authors who submitted such high-quality work. This conference is first and foremost a forum for disseminating, sharing, discussing, and debating world-class systems research. Thank you for your hard work and innovation! We thank the PC members and external reviewers for their significant investment of time, energy, and insight into shaping the program. We thank Pierre Tholoni and Roy Rinberg, who facilitated the PC meeting and made sure we ran the technology and not the other way around. We especially thank the USENIX staff, who have made chairing a conference like this one a well-oiled machine! Finally, we thank you for coming to this conference to engage with each other and with the authors of the accepted papers. This community is one we cherish, and one we are honored to have curated this year through our stewardship as co-chairs.

We hope that you leave this conference energized and inspired.

Onwards!

Roxana Geambasu, *Columbia University*

Ed Nightingale, *Apple*

OSDI '23 Program Co-Chairs

# 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)

July 10–12, 2023  
Boston, MA, USA

## Monday, July 10

### Make Your Bits Go Faster

- Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks** ..... 1  
Vishal Gupta, *EPFL*; Kumar Kartikeya Dwivedi, *SRMIST*; Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap, *EPFL*
- RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks** .....17  
Shiwu Lo, Han-Ting Lin, Yao-Hung Hsieh, and Chao-Ting Lin, *National Chung Cheng University*; Yu-Hsueh Fang, *National Cheng Kung University*; Ching-Shen Lin, *National Chung Cheng University*; Ching-Chun (Jim) Huang, *National Cheng Kung University*; Kam Yiu Lam, *City University of Hong Kong*; Yuan-Hao Chang, *Academia Sinica, Taiwan*
- Userspace Bypass: Accelerating Syscall-intensive Applications**..... 33  
Zhe Zhou, Yanxiang Bi, Junpeng Wan, and Yangfan Zhou, *Fudan University*; Zhou Li, *University of California, Irvine*
- Triangulating Python Performance Issues with SCALENE**..... 51  
Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno, *University of Massachusetts Amherst*
- Relational Debugging — Pinpointing Root Causes of Performance Problems** ..... 65  
Xiang (Jenny) Ren, Sitao Wang, Zhuqi Jin, David Lion, and Adrian Chiu, *University of Toronto*; Tianyin Xu, *University of Illinois at Urbana-Champaign*; Ding Yuan, *University of Toronto*

### Secure Your Bits I

- Accountable authentication with privacy protection: The Larch system for universal login** ..... 81  
Emma Dauterman, *UC Berkeley*; Danny Lin, *Woodinville High School*; Henry Corrigan-Gibbs, *MIT*; David Mazières, *Stanford University*
- K9db: Privacy-Compliant Storage For Web Applications By Construction**..... 99  
Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, and Malte Schwarzkopf, *Brown University*
- Encrypted Databases Made Secure Yet Maintainable** .....117  
Mingyu Li, *Shanghai Jiao Tong University*; *Shanghai AI Laboratory*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Xuyang Zhao and Le Chen, *Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Cheng Tan, *Northeastern University*; Huorong Li and Sheng Wang, *Alibaba Group*; Zeyu Mi, *Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Yubin Xia, *Shanghai Jiao Tong University*; *Shanghai AI Laboratory*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Feifei Li, *Alibaba Group*; Haibo Chen, *Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
- LVMT: An Efficient Authenticated Storage for Blockchain** ..... 135  
Chenxing Li, *Shanghai Tree-Graph Blockchain Research Institute*; Sidi Mohamed Beillahi, *University of Toronto*; Guang Yang and Ming Wu, *Shanghai Tree-Graph Blockchain Research Institute*; Wei Xu, *Tsinghua University*; Fan Long, *University of Toronto*
- Honeycomb: Secure and Efficient GPU Executions via Static Validation**..... 155  
Haohui Mai, *PrivacyCore Inc.*; Jiacheng Zhao, *SKLP, Institute of Computing Technology, CAS*; *Zhongguancun Laboratory*; and *UCAS*; Hongren Zheng, *IIS, Tsinghua University*; Yiyang Zhao, *SKLP, Institute of Computing Technology, CAS*; and *UCAS*; Zibin Liu, *BUPT*; Mingyu Gao, *IIS, Tsinghua University*; Cong Wang, *IDEA Shenzhen*; Huimin Cui, *SKLP, Institute of Computing Technology, CAS*; and *UCAS*; Xiaobing Feng, *SKLP, Institute of Computing Technology, CAS*; *Zhongguancun Laboratory*; and *UCAS*; Christos Kozyrakis, *PrivacyCore Inc. and Stanford*

## Secure Your Bits II

**An Extensible Orchestration and Protection Framework for Confidential Cloud Computing** .....173  
Adil Ahmad and Alex Schultz, *Arizona State University*; Byoungyoung Lee, *Seoul National University*; Pedro Fonseca, *Purdue University*

**Nimble: Rollback Protection for Confidential Cloud Services**..... 193  
Sebastian Angel, *Microsoft Research*; Aditya Basu, *Penn State University*; Weidong Cui, *Microsoft Research*;  
Trent Jaeger, *Penn State University*; Stella Lau, *MIT CSAIL*; Srinath Setty, *Microsoft Research*; Sudheesh Singanamalla, *University of Washington*

**Kerberos: Efficient and Scalable Cloud Admission Control** ..... 209  
Sultan Mahmud Sajal, *Microsoft Research and Pennsylvania State University*; Luke Marshall and Beibin Li, *Microsoft Research*; Shandan Zhou and Abhisek Pan, *Microsoft Azure*; Konstantina Mellou and Deepak Narayanan, *Microsoft Research*; Timothy Zhu, *Pennsylvania State University*; David Dion and Thomas Moscibroda, *Microsoft Azure*;  
Ishai Menache, *Microsoft Research*

**Security and Performance in the Delegated User-level Virtualization** ..... 227  
Jiahao Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Dingji Li, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; MoE Key Lab of Artificial Intelligence, AI Institute, *Shanghai Jiao Tong University*; Zeyu Mi, Yuxuan Liu, and Binyu Zang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*; Haibing Guan, *Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*; Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

**Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud**..... 247  
Ziqiao Zhou, *Microsoft Research*; Yizhou Shan, *University of California, San Diego*; Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann, *Microsoft Research*

## Tuesday, July 11

### Expanding, Hardening, and Deploying Your Bits

**ExoFlow: A Universal Workflow System for Exactly-Once DAGs**..... 269  
Siyuan Zhuang, *UC Berkeley*; Stephanie Wang, *UC Berkeley and Anyscale*; Eric Liang and Yi Cheng, *Anyscale*;  
Ion Stoica, *UC Berkeley*

**Hyrax: Fail-in-Place Server Operation in Cloud Platforms** ..... 287  
Jialun Lyu, *Microsoft Azure and University of Toronto*; Marisa You, Celine Irvine, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, and Savyasachi Samal, *Microsoft Azure*; Ioannis Manousakis and Lisa Hsu, *Formerly of Microsoft Azure*; Preetha Subbarayalu, Ashish Raniwala, Brijesh Warriar, and Ricardo Bianchini, *Microsoft Azure*;  
Bianca Schroeder, *University of Toronto*; Daniel S. Berger, *Microsoft Azure and University of Washington*

**NCC: Natural Concurrency Control for Strictly Serializable Databases by Avoiding the Timestamp-Inversion Pitfall** ..... 305  
Haonan Lu, *University at Buffalo*; Shuai Mu, *Stony Brook University*; Siddhartha Sen, *Microsoft Research*;  
Wyatt Lloyd, *Princeton University*

**Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta** ..... 325  
Boris Grubic, *Meta*; Yang Wang, *Meta and the Ohio State University*; Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, and Dan Kelley, *Meta*; Soteris Demetriou, *Meta and Imperial College London*; Kenny Yu and Chunqiang Tang, *Meta*

### Query Your Bits

**Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases** ..... 343  
Tamer Eldeeb and Xincheng Xie, *Columbia University*; Philip A. Bernstein, *Microsoft Research*; Asaf Cidon and Junfeng Yang, *Columbia University*



**ScaleDB: A Scalable, Asynchronous In-Memory Database** . . . . . 361  
Syed Akbar Mehdi, *The University of Texas at Austin*; Deukyeon Hwang and Simon Peter, *University of Washington*;  
Lorenzo Alvisi, *Cornell University*

**VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity** . . . . . 377  
Qianxi Zhang, Shuotao Xu, Qi Chen, and Guoxin Sui, *Microsoft Research Asia*; Jiadong Xie, *Microsoft Research Asia  
and East China Normal University*; Zhizhen Cai and Yaoqi Chen, *Microsoft Research Asia and University of Science and  
Technology of China*; Yinxuan He, *Microsoft Research Asia and Renmin University of China*; Yuqing Yang, Fan Yang,  
Mao Yang, and Lidong Zhou, *Microsoft Research Asia*

**Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction** . . . . . 397  
Zu-Ming Jiang and Si Liu, *ETH Zurich*; Manuel Rigger, *National University of Singapore*; Zhendong Su, *ETH Zurich*

**Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate** . . . . . 419  
Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, and Ion Stoica, *UC Berkeley*;  
Xiangyao Yu, *University of Wisconsin—Madison*

## Store Your Bits

**Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction** . . . . . 441  
Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu, *Tsinghua University*

**eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs** . . . . . 461  
Jaehong Min and Chenxingyu Zhao, *University of Washington*; Ming Liu, *University of Wisconsin-Madison*;  
Arvind Krishnamurthy, *University of Washington*

**SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory** . . . . . 479  
Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang, *The Chinese University of Hong Kong*

**No Provisioned Concurrency: Fast RDMA-codedigned Remote Fork for Serverless Computing** . . . . . 497  
Xingda Wei, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, and Shanghai AI  
Laboratory*; Fangming Lu, Tianxia Wang, Jinyu Gu, and Yuhan Yang, *Institute of Parallel and Distributed Systems,  
SEIEE, Shanghai Jiao Tong University*; Rong Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai  
Jiao Tong University, and Shanghai AI Laboratory*; Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE,  
Shanghai Jiao Tong University*

## Manage Your Bits I

**Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)**. . . . . 519  
Baptiste Lepers, *Université de Neuchâtel*; Willy Zwaenepoel, *University of Sydney*

**TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers**. . . . . 535  
Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee, *Stony Brook University*

**SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory** . . . . . 553  
Xuchuan Luo, *School of Computer Science, Fudan University*; Pengfei Zuo, *Huawei Cloud*; Jiacheng Shen and Jiazhen  
Gu, *The Chinese University of Hong Kong*; Xin Wang, *School of Computer Science, Fudan University*; Shanghai  
Key Laboratory of Intelligent Information Processing, Shanghai, China; Michael R. Lyu, *The Chinese University of  
Hong Kong*; Yangfan Zhou, *School of Computer Science, Fudan University*; Shanghai Key Laboratory of Intelligent  
Information Processing, Shanghai, China

**ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities** . . . . . 573  
Vasily A. Sartakov, Lluís Vilanova, and Munir Geden, *Imperial College London*; David Eyers, *University of Otago*;  
Takahiro Shinagawa, *The University of Tokyo*; Peter Pietzuch, *Imperial College London*

## Manage Your Bits II

**Global Capacity Management With Flux**. . . . . 589  
Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela  
Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and  
Chunqiang Tang, *Meta*

<b>Defcon: Preventing Overload with Graceful Feature Degradation</b> .....	<b>607</b>
Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard, <i>Meta Platforms, Inc.</i>	
<b>Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback</b> .....	<b>623</b>
Romil Bhardwaj, <i>UC Berkeley</i> ; Kirthevasan Kandasamy, <i>University of Wisconsin-Madison</i> ; Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica, <i>UC Berkeley</i>	
<b>Karma: Resource Allocation for Dynamic Demands</b> .....	<b>645</b>
Midhul Vuppapalati, Giannis Fikioris, and Rachit Agarwal, <i>Cornell University</i> ; Asaf Cidon, <i>Columbia University</i> ; Anurag Khandelwal, <i>Yale University</i> ; Éva Tardos, <i>Cornell University</i>	

## Wednesday, July 12

### Train Your Bits I

<b>AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving</b> .....	<b>663</b>
Zhuohan Li and Lianmin Zheng, <i>UC Berkeley</i> ; Yinmin Zhong, <i>Peking University</i> ; Vincent Liu, <i>University of Pennsylvania</i> ; Ying Sheng, <i>Stanford University</i> ; Xin Jin, <i>Peking University</i> ; Yanping Huang and Zhifeng Chen, <i>Google</i> ; Hao Zhang, <i>UC San Diego</i> ; Joseph E. Gonzalez and Ion Stoica, <i>UC Berkeley</i>	
<b>Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning</b> .....	<b>681</b>
Chen Zhang, <i>Tsinghua University</i> ; Lingxiao Ma and Jilong Xue, <i>Microsoft Research</i> ; Yining Shi, <i>Peking University &amp; Microsoft Research</i> ; Ziming Miao and Fan Yang, <i>Microsoft Research</i> ; Jidong Zhai, <i>Tsinghua University</i> ; Zhi Yang, <i>Peking University</i> ; Mao Yang, <i>Microsoft Research</i>	
<b>Welder: Scheduling Deep Learning Memory Access via Tile-graph.</b> .....	<b>701</b>
Yining Shi, <i>Peking University &amp; Microsoft Research</i> ; Zhi Yang, <i>Peking University</i> ; Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou, <i>Microsoft Research</i>	
<b>Effectively Scheduling Computational Graphs of Deep Neural Networks toward Their Domain-Specific Accelerators</b> .....	<b>719</b>
Jie Zhao, <i>Information Engineering University</i> ; Siyuan Feng, <i>Shanghai Jiao Tong University</i> ; Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyan Lv, and Qikai Xie, <i>Stream Computing Inc.</i>	

### Train Your Bits II

<b>EINNET: Optimizing Tensor Programs with Derivation-Based Transformations</b> .....	<b>739</b>
Liyang Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, and Shuhong Huang, <i>Tsinghua University</i> ; Xupeng Miao, <i>Carnegie Mellon University</i> ; Shizhi Tang and Kezhao Huang, <i>Tsinghua University</i> ; Zhihao Jia, <i>Carnegie Mellon University</i>	
<b>Hydro: Surrogate-Based Hyperparameter Tuning Service in Datacenters</b> .....	<b>757</b>
Qinghao Hu, <i>Nanyang Technological University, S-Lab, NTU, and Shanghai AI Laboratory</i> ; Zhisheng Ye, <i>Shanghai AI Laboratory and Peking University</i> ; Meng Zhang, <i>Nanyang Technological University, S-Lab, NTU, and Shanghai AI Laboratory</i> ; Qiaoling Chen, <i>Shanghai AI Laboratory and National University of Singapore</i> ; Peng Sun, <i>Shanghai AI Laboratory and SenseTime Research</i> ; Yonggang Wen and Tianwei Zhang, <i>Nanyang Technological University</i>	
<b>MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms</b> .....	<b>779</b>
Yuke Wang, Boyuan Feng, and Zheng Wang, <i>University of California Santa Barbara</i> ; Tong Geng, <i>University of Rochester</i> ; Kevin Barker and Ang Li, <i>Pacific Northwest National Laboratory</i> ; Yufei Ding, <i>University of California Santa Barbara</i>	
<b>Optimizing Dynamic Neural Networks with Brainstorm.</b> .....	<b>797</b>
Weihao Cui, <i>Shanghai Jiao Tong University</i> ; Zhenhua Han, <i>Microsoft Research Asia</i> ; Lingji Ouyang, <i>University of Science and Technology of China</i> ; Yichuan Wang, <i>Shanghai Jiao Tong University</i> ; Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, and Lidong Zhou, <i>Microsoft Research Asia</i> ; Quan Chen, <i>Shanghai Jiao Tong University</i> ; Haisheng Tan, <i>University of Science and Technology of China</i> ; Minyi Guo, <i>Shanghai Jiao Tong University</i>	

**AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models** . . . . .817  
Fan Lai, *University of Michigan*; Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, and Chun-cheng Jason Chen, *Meta*; Mosharaf Chowdhury, *University of Michigan*

## Verify Your Bits

**BWoS: Formally Verified Block-based Work Stealing for Parallel Processing** . . . . . 833  
Jiawei Wang, *Huawei Dresden Research Center, Huawei Central Software Institute, Technische Universität Dresden*; Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, and Jitang Lei, *Huawei Dresden Research Center, Huawei Central Software Institute*; Viktor Vafeiadis, *MPI-SWS*; Hermann Härtig, *Technische Universität Dresden*; Haibo Chen, *Huawei Central Software Institute, Shanghai Jiao Tong University*

**Spoq: Scaling Machine-Checkable Systems Verification in Coq** . . . . . 851  
Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh, *Columbia University*

**Verifying vMVCC, a high-performance transaction library using multi-version concurrency control** . . . . . 871  
Yun-Sheng Chang, *MIT CSAIL*; Ralf Jung, *ETH Zurich*; Upamanyu Sharma, *MIT CSAIL*; Joseph Tassarotti, *New York University*; M. Frans Kaashoek and Nickolai Zeldovich, *MIT CSAIL*

**Automated Verification of Idempotence for Stateful Serverless Applications** . . . . . 887  
Haoran Ding, Zhaoguo Wang, and Zhuohao Shen, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University*; Rong Chen, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University*; *Shanghai AI Laboratory*; Haibo Chen, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University*

**Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems** . . . . . 911  
Travis Hance and Yi Zhou, *Carnegie Mellon University*; Andrea Lattuada, *VMware Research*; Reto Achermann, *University of British Columbia*; Alex Conway, *VMware Research*; Ryan Stutsman, *VMware Research and University of Utah*; Gerd Zellweger, *VMware Research*; Chris Hawblitzel, *Microsoft Research*; Jon Howell, *VMware Research*; Bryan Parno, *Carnegie Mellon University*

## Transfer Your Bits

**Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs** . . . . . 931  
Qiang Li, *Alibaba Group*; Yixiao Gao and Xiaoliang Wang, *Nanjing University*; Haonan Qiu, *Alibaba Group*; Yanfang Le, *AMD*; Derui Liu, *Alibaba Group*; Qiao Xiang, *Xiamen University*; Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, and Zhongjie Wu, *Alibaba Group*; Chen Tian and Guihai Chen, *Nanjing University*; Dennis Cai, Jinbo Wu, Jiayi Zhu and Jiesheng Wu, *Alibaba Group*; Jiwu Shu, *Xiamen University*

**ShRing: Networking with Shared Receive Rings** . . . . . 949  
Boris Pismenny, *Technion & NVIDIA*; Adam Morrison, *Tel Aviv University*; Dan Tsafir, *Technion & VMware Research*

**ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta** . . . . . 969  
Harshit Saokar, *Meta*; Soteris Demetriou, *Meta and Imperial College London*; Nick Magerko, Max Kontorovich, Josh Kirstein, and Margot Leibold, *Meta*; Dimitrios Skarlatos, *Meta and Carnegie Mellon University*; Hitesh Khandelwal and Chunqiang Tang, *Meta*

**Characterizing Off-path SmartNIC for Accelerating Distributed Systems** . . . . . 987  
Xingda Wei and Rongxin Cheng, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University, and Shanghai AI Laboratory*; Yuhan Yang, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University*; Rong Chen, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University, and Shanghai AI Laboratory*; Haibo Chen, *Institute of Parallel and Distributed Systems, SEITEE, Shanghai Jiao Tong University*

**Ensō : A Streaming Interface for NIC-Application Communication** . . . . . 1005  
Hugo Sadok and Nirav Atre, *Carnegie Mellon University*; Zhipeng Zhao, *Microsoft*; Daniel S. Berger, *Microsoft Research and University of Washington*; James C. Hoe, *Carnegie Mellon University*; Aurojit Panda, *New York University*; Justine Sherry, *Carnegie Mellon University*; Ren Wang, *Intel*



# Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks

Vishal Gupta Kumar Kartikeya Dwivedi\* Yugesh Kothari Yueyang Pan

Diyu Zhou Sanidhya Kashyap

EPFL \*SRMIST

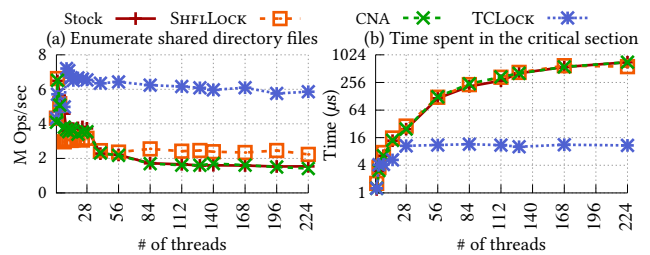
## Abstract

Today’s high-performance applications heavily rely on various synchronization mechanisms, such as locks. While locks ensure mutual exclusion of shared data, their design impacts application scalability. Locks, as used in practice, move the lock-guarded shared data to the core holding it, which leads to shared data transfer among cores. This design adds unavoidable critical path latency leading to performance scalability issues. Meanwhile, some locks avoid this shared data movement by localizing the access to shared data on one core, and shipping the critical section to that specific core. However, such locks require modifying applications to explicitly package the critical section, which makes it virtually infeasible for complicated applications with large code bases, such as the Linux kernel.

We propose *transparent delegation*, in which a waiter automatically encodes its critical section information on its stack and notifies the combiner (lock holder). The combiner executes the shipped critical section on the waiter’s behalf using a lightweight context switch. Using transparent delegation, we design a family of locking protocols, called TCLocks, that requires zero modification to applications’ logic. The evaluation shows that TCLocks provide up to  $5.2\times$  performance improvement compared with recent locking algorithms.

## 1 Introduction

Synchronization mechanisms are the basic building blocks for today’s high-performance concurrent applications. In fact, applications heavily rely on locks as a concurrency control mechanism, as they provide a set of simple programming APIs for users to mediate concurrent access to shared data. Besides ensuring program correctness, locks also affect the scalability of applications [33, 34, 49]. For instance, various high-performance applications, such as the Linux kernel, have moved from coarse-grained to fine-grained locks [52] for minimizing the length of the critical section. However, thanks to diverse workloads and applications, the scalability problem due to lock algorithms still remains at large [41, 55, 62, 70].



**Figure 1:** Impact of locks on a file-system micro-benchmark [62]. We compare three traditional lock algorithms: Linux’s `qspinlock` (*Stock*) [41], `CNA` [43] and `SHFLOCK` [52] with our proposed `TCLock`. (a) Enumerating files in a shared directory on an 8-socket 224-core machine. (b) Time spent in the critical section: moving the critical section context (`TCLock`) compared with moving critical section shared data (*Stock*, `CNA`, and `SHFLOCK`).

As a result, research in lock algorithms focuses on minimizing the contention on cache-lines containing the lock word and shared data. The most widely used algorithms always move the shared data to the core executing the critical section [30, 41, 43, 52, 53, 60]. Lock evolution within this design philosophy has focused on reducing contention on the lock word. However, such a lock design still moves shared data across cores for every lock acquisition. **Figure 1** shows that such shared data access cost increases with increasing cores, thereby limiting the scalability of applications.

On the other end of the design spectrum, some algorithms adopt the request-response style of communication, also called delegation-style locking [37, 47, 51, 56, 64, 67]. Specifically, waiters delegate their critical section execution context to a dedicated core [56, 67] or a *combiner* [47, 51] that executes that function on behalf of each waiter in a specific order. **Figure 1** illustrates that this design outperforms traditional locks and improves application performance. In particular, such a lock design minimizes the shared data movement, and ensures almost constant critical section latency regardless of the number of threads.

Despite potential performance gains, the practical design and implementation of delegation-style locks faces several challenges. First, applications require major rewriting to ex-

explicitly encapsulate and pass the critical section as a function pointer [51, 56, 67]. Unfortunately, this rewriting becomes impractical for applications with large code bases, such as the Linux kernel, which has over 180k lock API call sites [52]. Second, every delegation-based work focuses on situations involving a single lock contention. However, today's applications often employ fine-grained locking and may acquire multiple locks for operations, such as memory, scheduler, and storage management in the Linux kernel [4, 12, 13, 20]. Finally, the third challenge involves managing the per-CPU or per-thread variables, which applications heavily depend on for either performance or correctness.

In this paper, we take the first step towards making delegation-based locks practical for concurrent applications with large code bases. We introduce the idea of *transparent delegation*, which enables developers to utilize delegation-style locking without rewriting the application. Our transparent delegation approach encapsulates the critical section using two observations: First, a thread's stack and CPU registers contain the state of the waiter's thread. Second, using the lock/unlock API pushes the thread's context on its stack. Thus, a waiter *saves its critical section context using CPU registers and stack pointer, and calling the lock API as a function*. Finally, the combiner executes the waiter's critical section on its behalf by assuming the role of the waiter using a lightweight context switch mechanism [29, 59]. This context-switch mechanism is transparent to the application.

Using transparent delegation, we design a new family of locks called TLocks that augment existing locks, such as test-and-set (TAS) and MCS, by employing the combining technique for batching waiters' requests [47]. Our first lock is a spinlock, where waiters continuously spin while awaiting the lock. The combiner can execute multiple waiter's critical section before passing its role based on a counter-based mechanism. Similar to our prior work (SHFLocks) [52], our algorithms can enforce hardware and software policies on the fly. In particular, our spinlock version also incorporates NUMA-awareness policy. We then integrate the core over-subscription policy [52] to design a blocking lock, where the waiter can sleep while waiting for the lock. Lastly, we design a phase-based readers-writer lock built on top of our blocking lock.

Applying TLocks directly in highly concurrent systems presents its own set of challenges. First, transparent delegation violates the single-writer property of a thread's stack, meaning that two threads (the combiner and the waiter) writing to the same stack can cause data races and stack corruption. Waiters need access to a stack due to specific events, such as interrupts in the kernel space, signals in userspace, and scheduling of waiting threads. We address the data-race issue using a *per-thread ephemeral stack* that a waiter switches to between the acquire and release phases.

Second, most concurrent applications use multi-level locking [4, 20, 28] and out-of-order (OOO) unlocking [12, 13]

for higher concurrency and better scalability. TLocks handle the arbitrary level of nested combining by maintaining combiner-specific state on the ephemeral stack before acquiring the nested lock. Meanwhile, we handle OOO unlocking by keeping track of the order of acquired locks. We delay the release of OOO unlocked locks until the order is the inverse of acquired locks. This effectively flattens the release of locks.

We evaluate TLocks in both kernel space and userspace on NUMA machines. TLocks improve the performance within and across sockets. Specifically, TLocks boost application throughput by 1.7–5.2× compared to the locks used in the Linux kernel and state-of-the-art locks, respectively.

In summary, this paper makes the following contributions:

- **Design technique.** We introduce a new design technique called transparent delegation. Locks with this technique allow developers to use the same APIs as traditional locks while benefiting from the scalability improvements provided by delegation-style locking.
- **Delegation-based lock family.** We implement TLocks that employ transparent delegation. We first design a spinning lock and extend it to blocking and readers-writer locks, utilizing per-thread ephemeral stacks to manage the parking of waiters.
- **Practical application.** TLocks incorporate various lock use scenarios, including nested locking and out-of-order unlocking. This approach allows us to realize the potential of delegation-style locking for the Linux kernel without modifying any code.

## 2 Background

While executing a critical section, a thread accesses three types of memory locations (data):

1. **Lock word**, *i.e.*, its structure that determines the exclusive access for a thread.
2. **Shared data** among threads guarded by a lock word, accessible only to the thread holding the lock.
3. **Thread-local data** like stack and per-thread variables.

Most lock designs minimize the contention on the lock word, while some minimize the movement of shared data. Hence, there are two design philosophies based on shared data movement: traditional and delegation-style. We now discuss the evolution of locks based on these design philosophies. Later, we touch upon the systems-level challenges that are specific to delegation-style locks.

### 2.1 Traditional Locks

Traditional lock design adheres to the principle of moving data to computation. A core executes the critical section by moving shared data into its cache. Consequently, this design moves cache lines of both the lock word and shared data across cores while executing the critical section. The evolution of traditional lock algorithms [50] has focused on minimizing cache-line movement of the lock word. For example, queue-based locks [41, 42, 44, 58, 60, 68] minimize

cache-line contention due to the lock word. Hierarchical locks [39, 46, 57, 66] further reduce the cache-line contention on non-uniform memory access (NUMA) machines, where accessing a local-socket memory location is faster than a remote one. These locks amortize the remote access cost of the lock word by reordering the wait queue to pass the lock within the same socket. SHFLLOCK [52] and CNA [43] further generalize hierarchical lock design by reordering the wait queue based on various hardware and software policies. Moreover, our recent work [65] has also shown that the reordering policy can be changed dynamically without kernel compilation.

Readers-writer locks also follow traditional lock design, with most locks aiming to minimize contention on the lock word [36, 54, 61]. These locks augment mutually exclusive locks with different types of read indicators based on workload requirements. Some examples include centralized [61], per-socket [38], and per-CPU [40, 63, 71] indicators. These locks also require moving shared data across cores, even though they offer a broader semantics of mutual exclusion.

Traditional locks do not require modifying applications since the lock/unlock programming APIs remain consistent. However, these locks move shared data cache lines among cores while executing the critical section. Unfortunately, this lock design incurs shared data movement for every critical section execution, thereby increasing critical section execution latency. Moreover, this latency grows with increasing core count (Figure 1 (b)), which saturates the throughput without efficiently utilizing hardware.

## 2.2 Delegation-style Locks

Delegation-style locks follow the principle of moving computation to data [45, 47, 51, 56, 64, 67]. These locks use an old technique called combining that has been used in hardware and software to mitigate memory contention by combining requests for the same memory location. In this approach, waiters pack their critical section as a function and pass that function pointer to the combiner as a request. The combiner then executes the waiter's function and notifies it upon completion. Executing the critical section on the same core eliminates shared data movement, leading to improved application throughput with increasing core count (Figure 1).

However, this lock design has a critical limitation. It does not provide the same lock/unlock APIs as traditional locks [47, 67]. Consequently, we need to modify applications, which involves identifying each critical section in the code, wrapping it as a function, and modularizing the application logic for delegation. Modifying application logic to encapsulate the critical section as a function is quite challenging and even impossible in some cases [46]. For instance, Roghanchi *et al.* [67] reported modifying  $\sim 1,500$  lines of code (LoC) to enable delegation for Memcached. This limitation, unfortunately, prevents the scalability offered by delegation-style locking from being applied to existing real-world applica-

tions, such as Linux, which comprises 28M LoC with more than 180k static lock call sites.

## 2.3 The Incompatibility of Delegation in Concurrent Applications

Real-world applications, such as the Linux kernel, employ fine-grained locking in multiple execution contexts. Fine-grained locking mostly involves acquiring multiple nested locks when working with several objects. To prevent deadlocks, the nested locks are acquired in a specific order, but they can be released in arbitrary order to enhance concurrency [12, 13]. However, none of the delegation approaches handle such common, but challenging cases. We measured that both nested locking calls and OOO unlocking calls are quite prevalent. For instance, booting Linux results in  $\sim 80k$  nested locking calls and  $\sim 20k$  OOO unlock calls. Thus, addressing these cases is essential to make delegation-style locks practical for every concurrent systems software.

In addition, the Linux kernel can call locks from various contexts. These contexts comprise of task [25] and interrupts (*e.g.*, non-maskable interrupt context [11], HardIRQ context [6, 7], or SoftIRQ context [23]). The kernel code typically executes in the task context. Depending on the kernel configuration, a scheduler can preempt or migrate a task to another CPU. Nevertheless, in special execution contexts, such as interrupts, or code regions that disable CPU preemption and migration, the scheduler prohibits the migration of such contexts. The Linux kernel code also heavily utilizes per-CPU variables and implicitly relies on stable access to these variables in such special execution contexts. Traditional lock design does not require any handling for special execution contexts because the critical section executes on the core that acquires the lock. In contrast, delegation-style locks break this property, necessitating special handling for these cases, *i.e.*, special contexts and stable access to per-CPU variables.

**Goal.** In accordance with the general design principle of minimizing data movement [43, 52], our objective is to reduce data movement for both lock word and shared data. Unlike existing delegation-style locks, we avoid making any code modifications. Hence, we take the initial step towards achieving the goal of transparently enabling delegation-style locking for any real-world application, including the OS.

## 3 TLocks

We propose transparent delegation, a practical lock-design technique for real-world applications that allow developers to use the same lock/unlock APIs as traditional locks without modifying the application code. Transparent delegation involves two steps: First, it automatically encapsulates a critical section of arbitrary length in a set of registers and the thread stack. Second, waiters pass this encapsulated information to the combiner for execution. As a result, our approach enables applications to enjoy scalability without any modification. We apply this technique to design a family



of lock algorithms called TLocks, that transparently delegate waiters' requests to the combiner. TLocks comprises spinning (§3.2) and blocking (§3.4) locks. We extend the blocking lock with read indicators to design a phase-based readers-writer blocking lock (§3.5) incorporating hardware and software-based optimizations (§3.6).

### 3.1 TLock Design

We first discuss a set of insights and techniques that allows us to design and implement TLocks.

**Transparent delegation.** When executing a critical section, a thread can access both shared data (e.g., global variables, heap) and thread-local data (e.g., registers, stack, and per-thread variables). In delegation-style locking, although shared data is globally available to all threads, the combiner requires access to the waiter's thread-local data and the set of instructions for executing its critical section.

Our technique overcomes the challenge of thread-local data and critical section context using three key insights.

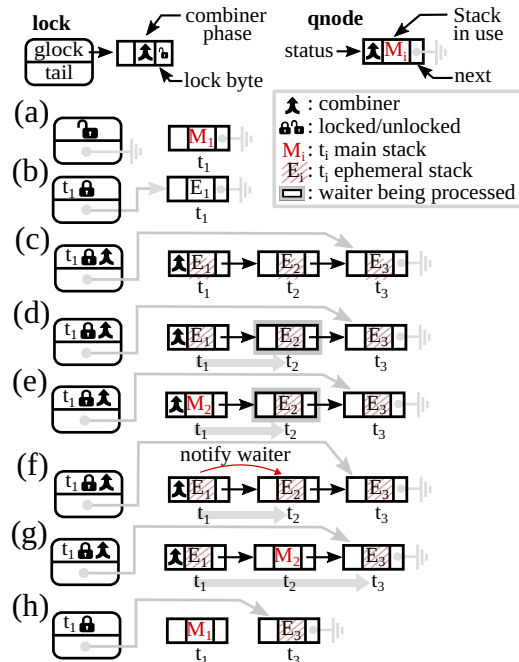
1. A thread's execution context is well-defined by hardware, with thread-specific CPU registers and the stack containing all information for executing the critical section [2, 8, 10].
2. A waiter busy-waits without modifying its state once it sends its request to the combiner. It exits only after receiving the response from the combiner.
3. Calling the lock API as a function<sup>1</sup> ensures that hardware pushes the next instruction onto the stack, making the critical section's start address available to the combiner for executing the critical section.

Using these insights, the combiner pops the start address of a critical section from the waiter's stack using a *return* instruction and executes it. After completing the critical section, calling the unlock API pushes the first instruction of the non-critical section onto the waiter's stack. The waiter resumes executing the non-critical section after receiving a notification from the combiner. Thus, transparent delegation allows waiters to seamlessly pass context and resume after the critical section's execution.

**Avoiding concurrent stack access with an ephemeral stack.** To ensure program correctness, transparent delegation must prevent concurrent accesses to a thread's execution stack. Ideally, a waiter busy-waits for notification during its critical section execution. However, certain events, such as interrupts in kernel space, signals in user space, and the waiter's parking and wake-up mechanism [32] can access the waiter's stack during the execution of its delegated critical section. As a result, naive transparent delegation via stack switching violates the fundamental single-writer stack principle, leading to potential stack state corruption.

To address this issue, we introduce an initially empty, separate stack called the *ephemeral stack*. Each waiter switches to its ephemeral stack during lock acquisition, and delegates

<sup>1</sup>For example, the `call` instruction in x86.



**Figure 2:** Lock and qnode structures of the TLock. (a) Initially, the lock is in the unlock state.  $t_1$  first switches from its main stack ( $M_1$ ) to the ephemeral stack ( $E_1$ ), and (b) joins the waiting queue. (c)  $t_1$  being at the head of the queue, becomes the combiner. Meanwhile,  $t_2$  and  $t_3$  join the queue. They also switch their stack to  $E_2$  and  $E_3$ , respectively. (d)  $t_1$  begins the combining process by traversing the queue and finds  $t_2$ . (e)  $t_1$  switches to  $t_2$ 's main stack ( $E_1 \rightarrow M_2$ ) and executes  $t_2$ 's critical section. (f) Once finished,  $t_1$  first switches back to  $E_1$  and then notifies  $t_2$  that  $t_1$  has finished executing its critical section. (g)  $t_2$  then switches back to  $M_2$  and exits its unlock phase. Meanwhile,  $t_1$  finds  $t_3$  as the last waiter. (h)  $t_1$  notifies  $t_3$  that it is now at the head of the queue, then  $t_1$  switches its stack to  $M_1$ , executes its critical section, and finally exits the unlock phase.

its critical section to the combiner. The waiter then busy-waits using the ephemeral stack while the combiner accesses the waiter's main stack to execute the critical section. Importantly, the use of an ephemeral stack does not introduce any new stack overflow bugs since it is a separate memory from the thread's execution stack. By incorporating the ephemeral stack, TLocks maintain the single-writer principle, thereby preventing concurrent access and the corruption of waiters' stack.

### 3.2 Spinlock: TLock<sup>SP</sup>

TLock<sup>SP</sup> augments the TAS and MCS lock by adopting the combining technique from MCS-style combining works [45, 47]. It involves a waiting thread becoming a combiner and batch waiters' requests up to a set threshold. Specifically, TLock extends the DSM-Synch lock, using TAS as a top-level lock, and an MCS-style waiting queue for waiters. The waiter's queue node (qnode) maintains additional states: 1) request and wait flags for synchronizing between a waiter and the combiner and selecting the next combiner. 2) Socket ID for NUMA lock design (§3.6.3). 3) Batch count to limit

excessive waiters, causing starvation or long-term fairness issues. And, 4) a pointer to the waiter's thread context for transparent delegation, which includes all registers and the stack pointer.

**Transparent delegation invariants.** Our lock algorithm maintains four invariants: 1) A combiner is always at the head of the waiting queue. 2) A waiter never uses its main stack while busy waiting. 3) All instructions in a critical section are executed only once, either by a waiter or the combiner executing on the waiter's behalf. 4) A combiner exclusively executes the waiter's critical section instructions defined between the lock and the unlock phase.

**Workflow.** Figure 2 presents a running example of  $\text{TCLock}^{SP}$ . Before requesting a lock, every thread executes in its main context (a). When a thread requests a lock, it switches to an ephemeral stack, saves its main context in its `qnode`, and joins the queue (b). Now, the head of the queue ( $t_1$ ) becomes the combiner, while other threads ( $t_2$  and  $t_3$ ) join the queue after switching to their respective ephemeral stacks. They wait for notification from the combiner, while processing any interrupts and signals on their ephemeral stacks. The combiner iterates through the queue and finds  $t_2$ 's request (d).  $t_1$  context-switches to  $t_2$ 's main context using  $t_2$ 's `qnode`, and starts executing  $t_2$ 's critical section (e). Reaching the unlock API of  $t_2$ ,  $t_1$  switches back to its ephemeral stack, notifies  $t_2$ , and checks for other requests (f). Once  $t_2$  receives notification, it switches back to its main context, which now points to the end of the critical section. It then continues executing its non-critical section (g). Finally, the combiner iterates through the entire queue, it passes the combining role to  $t_3$ , switches to its main context, and executes its critical section (h). Finally,  $t_1$  unlocks the lock, allowing  $t_3$  to acquire it and continue the combining process.

**Algorithm.** Listing 1 presents the  $\text{TCLock}^{SP}$  pseudocode. A thread  $t$  first attempts to acquire the TAS lock on the fast path (line 17). On success,  $t$  executes its critical section directly. Otherwise,  $t$  finds its thread-local combining structure (line 21), saves its register state on the main stack, switches to the ephemeral stack, and begins the slow path (lines 26–27). The slow path comprises four phases: 1)  $t$  joins the queue and busy-waits locally. 2)  $t$  then waits to acquire the TAS lock after becoming the head of the queue. 3) After acquiring the TAS lock,  $t$  checks the combining conditions. 4) Finally,  $t$  combines waiters' critical sections.

*Phase 1: Busy-waiting phase.* Upon entering the slow path,  $t$  initializes its `qnode` (line 32). Specifically, it sets the `wait` field to `True`, `request` field to `UNPRCSD`, and the `next` pointer to `None`. The combiner notifies a waiter with the `wait` flag and uses the `request` flag to specify whether it executed a waiter's critical section.  $t$  then adds itself to the waiting queue by atomically swapping the `tail` with the `qnode`'s address (line 36). After that,  $t$  checks for any preceding waiters

in the queue. If true,  $t$  joins the queue as a waiter (line 38) and waits for the combiner's notification (lines 39–40); otherwise, it proceeds to phase 2. While in the queue,  $t$  busy-waits for the combiner to execute its critical section. After reaching the end of  $t$ 's critical section, the combiner pushes the first instruction after the unlock API (line 107) on  $t$ 's main stack. It then marks  $t$ 's request as complete (*i.e.*, `PRCSD`) (line 28), which switches to its main context and begins the non-critical section (line 107). If, however,  $t$ 's request is not completed,  $t$  reaches the head of the queue and moves to phase 2.

*Phase 2: Global lock acquisition phase.*  $t$  now tries to acquire the TAS (global) lock using the CAS operation (lines 46–50).

*Phase 3: Combining-role decision phase.* After acquiring the global lock,  $t$  checks whether it can be a combiner (lines 52–56). If  $t$  is the only one in the queue, it resets the queue tail, (lines 52–53), switches to its main stack (line 28), executes its critical section, and releases the lock. Otherwise,  $t$  checks if there are at least two waiters in the queue. If true,  $t$  proceeds to phase 4 as a combiner. Otherwise,  $t$  passes the combining role to the next waiter (lines 58–60) by setting the `wait` bit to false, and releases the lock after executing its critical section.

*Phase 4: Combining phase.*  $t$  begins the combining phase by disabling the fast path, thereby forcing new waiters to join the queue (line 63).  $t$  iterates over the queue to execute each waiter's critical section (lines 65–75). Within the loop,  $t$  selects the next waiter (4a line 67), and records the waiter's information (`qnode`) in its thread-local combiner struct (`cst`) to later use it for resuming the combining process. Then  $t$  switches from its ephemeral stack to the waiter's main stack, and executes the waiter's critical section (4b line 70). Once finished,  $t$  notifies the waiter by first setting the `request` flag to `PRCSD` and resetting the `wait` flag (4c line 72).  $t$  continuously iterates until it reaches the combining threshold or cannot find two subsequent waiters in the queue (4d lines 73–75). After exiting the loop,  $t$  ends the combining phase by changing the locking mode to the non-combining mode (line 77).  $t$  then notifies the next waiter to be the head of the queue (line 78), and finally executes its critical section.

During the unlock phase,  $t$  can be in one of the two states: `G_LOCKED`:  $t$  unlocks the TAS lock by resetting its value and returns (lines 92–94). `G_LOCKED_COMBINER`:  $t$  does not release the TAS lock. It context switches from a waiter to the combiner by switching from the waiter's main stack to the combiner's ephemeral stack (lines 98) After switching,  $t$  resumes the combining loop and notifies the waiter about the completion of the critical section (line 72).

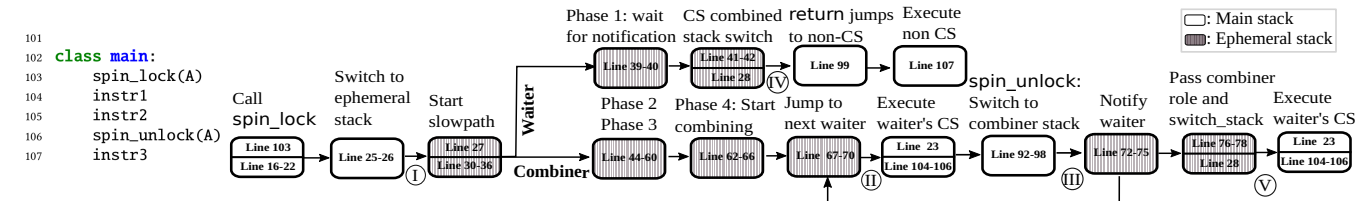
### 3.3 Proof Sketch of Correctness

**Mutual exclusion.**  $\text{TCLock}$  ensures mutual exclusion by maintaining two invariants: First, only one thread can hold the global TAS lock, which can also be a combiner; Second, the main stack of a thread is active on only one thread at any time.  $\text{TCLock}$  piggybacks on the mutual exclusion property

```

1  PRCSDB = 0 # Waiter's request is processed by the combiner
2  UNPRCSDB = 1 # Waiter's request is not processed until now
3  G_UNLOCK = 0, G_LOCKED = 1 # TAS known states
4  G_LOCKED_COMBINER = 2 # State to mark combining phase
5  WAITERS_TO_COMBINE = 1024 # Combining batch count
6
7  class thread_local_combiner_struct:
8      qcurr = None, qprev = None, qnext = None, node = init_node()
9      counter = 0, lock_addr = Array[None]
10     estack_rsp = init_ephemeral_stack()
11
12  class lock:
13     glock = 0, tail = None # TAS: top level lock, MCS queue
14
15  def spin_lock(lock):
16     # Fastpath: Try to acquire the TAS lock
17     if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
18         return # Got the lock, going to execute the critical section
19
20     # Switch to the ephemeral stack and acquire the lock in slowpath
21     cst = this_thread_comb_struct() # Get the per-CPU combiner struct
22     switch_stack(lock, cst) # Switch stack and begin slowpath function
23     return
24
25  def switch_stack(lock, cst):
26     switch_to_ephemeral_stack(cst.node) # Main → ephemeral stack
27     lock_slowpath(lock, cst)
28     switch_from_ephemeral_stack(cst.node) # Ephemeral → main stack
29
30  def lock_slowpath(lock, cst):
31     qnode = cst.node # Get the pointer to qnode in the combiner struct
32     init_qnode(qnode, wait = True, request = UNPRCSDB,
33               next = None, skt_id = numa_id()) # Initialize waiter's qnode
34
35     # Phase 1: Busy waiting: Join the queue and wait until notified
36     qprev = SWAP(&lock.tail, &qnode) # Atomically add node to tail
37     if qprev is not None: # Waiters are already present in the queue
38         qprev.next = qnode # Link qprev with qnode to form a queue
39         while qnode.wait is True:
40             continue # Wait for the combiner to halt waiter's spinning
41         if qnode.request == PRCSDB: # Waiter request has been processed
42             return #Combiner executed my CS; jump to non critical section
43
44     # Phase 2: Global lock acquisition: Acquire the TAS lock
45     # Waiter is at the head of the queue; get the TAS lock
46     while True: # Wait for the glock to be unlocked
47         while lock.glock != G_UNLOCK:
48             continue
49         if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
50             break # Got the TAS lock
51
52     # Phase 3: Combining-role decision: Whether to combine
53     if CAS(&lock.tail, qnode, None) == qnode:
54         return # If only one in the queue, return
55     qnext = qnode.next # Someone joined the queue; get qnode ptr
56     while qnext is None:
57         qnext = qnode.next
58     # If there are at least two waiters, start combining
59     if qnext.next == None:
60         notify_next_queue_head(qnext) # next waiter is combiner
61         return
62
63     # Phase 4: Combining: Batch requests with dynamic policies
64     lock.glock = G_LOCKED_COMBINER # Declare combining phase
65     counter = 0
66     while True: # Combiner loop
67         qcurr = qnext # Get the very next waiter after combiner
68         qnext = select_next_waiter(qcurr) # 4a Get the next node
69         cst.qcurr = qcurr # For qcurr's stack switch in unlock()
70         # 4b Combiner's ephemeral stack → next waiter's stack
71         switch_stack_from_combiner_to_waiter(cst, qcurr)
72         # Waiter's critical section execution finished
73         notify_waiter(qnode) # 4c Mark as completed
74         if qnext is None or qnext.next is None or
75             counter >= WAITERS_TO_COMBINE: # 4d Check comb. cond.
76             break
77     # Combiner phase is over, now combiner runs its CS
78     lock.glock = G_LOCKED # Reset TAS lock to normal lock
79     notify_next_queue_head(qnext) # Next waiter is the combiner
80
81     # Select the next node based on the policy, eg., NUMA etc.
82     def select_next_waiter(qnode):
83         return qnode.next
84
85     def notify_next_queue_head(qnode):
86         qnode.wait = False
87
88     def notify_waiter(qnode):
89         qnode.request = PRCSDB
90         qnode.wait = False
91
92     def spin_unlock(lock):
93         if lock.glock == G_LOCKED:
94             lock.glock = G_UNLOCK # Only true for no combining phase
95             return
96         # Jump back to combiner
97         cst = this_cpu_comb_struct()
98         # Waiter's stack → combiner's ephemeral stack
99         switch_stack_from_waiter_to_combiner(cst.qcurr, cst)
100        return

```



**Listing 1:** Pseudocode of TLock along with the algorithm flow. In the bottom figure, Shade of the boxes show which stack is currently active and the numbers ①–⑤ shows stack switching locations in the algorithm. At each of these locations, following return addresses are present on the outgoing stack: ① Outgoing: Waiters’ main stack → line 23. ② Outgoing: Combiner’s ephemeral stack → line 71. ③ Outgoing: Waiter’s main stack → line 99. ④ and ⑤ Outgoing: waiter’s and combiner’s ephemeral stack → contents are discarded.

of the TAS lock as it uses atomic compare-and-swap (CAS) to guarantee thread exclusivity. Hence, thanks to the TAS lock, only one thread can hold the global lock at any given point and only one thread can access shared data at a time. Finally, our transparent delegation invariants ensure that a waiter never touches its own main stack while waiting for a combiner’s notification. Moreover, TLock ensures that

the switch from the waiter’s main stack to the combiner’s ephemeral stack (line 98) occurs at the end of the critical section, i.e., at the end of the unlock function. Thus, after the waiter restores its context from the main stack (line 28), it never enters its critical section.

**Correct thread state.** TLock preserves the correct waiter’s state using a lightweight context switch mechanism



and avoids concurrent stack modification. Specifically, a waiter yields ownership of its main stack (line 26) before joining the waiting queue (line 36). Thus, a combiner thread can only obtain the ownership of a waiter thread's main stack after the waiter gives up the ownership. Finally, the combiner thread concedes its ownership of the waiter's main stack (line 98) before notifying the waiter (line 72). Therefore, our approach ensures that when a waiter reacquires the ownership of its main stack (line 28), the combiner is not using that stack.

### 3.4 Blocking Lock: $\text{TCLock}^B$

$\text{TCLock}^B$  follows a similar design philosophy of the blocking  $\text{SHFLock}$ , where waiters use the spin-then-park strategy. In this approach, a waiter spins locally until its time quota expires. Upon expiration, it schedules itself out if the system is oversubscribed; otherwise, it yields to the scheduler, which eventually reschedules the waiter. In addition, the lock queue maintains both active and passive waiters.

We design  $\text{TCLock}^B$  by augmenting  $\text{TCLock}^{SP}$  to support the parking/wakeup policy. We extend the combiner's role, which now wakes up sleeping waiters while executing their critical sections. The use of an ephemeral stack becomes critical for  $\text{TCLock}^B$  because parking of waiters requires calling a function, which pushes the function frame on the waiter's stack. Hence,  $\text{TCLock}^B$  uses the thread-local ephemeral stack to prevent concurrent accesses. The stack switching protocol remains the same as in  $\text{TCLock}^{SP}$ . To enable efficient parking and wakeup, we add two new states to the request field of the  $qnode$ : `PARKED`, in which a waiter is scheduled out, and `PRCSING`, which indicates that the combiner has started executing a waiter's critical section.

In the slow-path phase, while spinning locally (*i.e.*, phase 1), a waiter  $t$  checks if its time quota is up. If so,  $t$  attempts to park itself out. Specifically,  $t$  tries to change its request field from `UNPRCSD` to `PARKED` atomically. If successful,  $t$  parks itself out; otherwise, it continues spinning as the combiner has changed  $t$ 's state. In phase 4c, while selecting the next head of the queue (`notify_next_queue_head()`), the combiner atomically swaps  $t$ 's state to `PRCSING` to prevent the waiter from going to sleep. Furthermore, after executing the critical section, the combiner atomically swaps  $t$ 's state to `PRCSD`. In both cases, the combiner checks the old state of the request field. If it is `PARKED`, the combiner wakes up  $t$ . We use atomic instructions for changing the state to prevent the lost wakeup problem.

### 3.5 Readers-writer Version: $\text{TCLock}^{RW}$

$\text{TCLock}^{RW}$  is a combining-aware readers-writer lock that allows readers to execute in parallel, while writers are combined.  $\text{TCLock}$  uses a phase-based mechanism [35, 36] that alternates between readers and combined writers.  $\text{TCLock}^{RW}$  comprises the following: 1) A counter that includes the reader count (`RCNT`), writer present byte (`WP`) to indicate if a writer is holding the lock, and writer waiting

```

1 RCNT = 1 << 16; WW = 0x100; WP = 0x1;
2 WCOMBINER = G_LOCKED_COMBINER
3 class rwlock: (32 byte lock)
4   # rwcounter → [RCNT: 17-63; WW: 8-16; WP: 0-7]
5   rwcounter = 0 # 8-byte readers-writer rwcounter
6   tail = None # Writers enqueue in this queue
7   wlock: mutex # Coordinate bw readers & first writer
8
9 def down_read(rwlock): # Acquire read lock
10  atomic_inc(&rwlock.rwcounter, RCNT) # Increment reader count
11  if !(rwlock.rwcounter & 0xffff): # Check the first two bytes
12    return # Lock acquired, if writer not present or waiting
13  atomic_dec(&rwlock.rwcounter, RCNT) # Decrement reader count
14  read_lock_slowpath(rwlock) # execute read slowpath
15
16 def read_lock_slowpath(rwlock):
17  mutex_lock(&rwlock.wlock) # Acquire mutex
18  atomic_inc(&rwlock.rwcounter, RCNT) # Increment reader count
19  while (rwlock.rwcounter & 0xffff) > 0: # Check first two bytes
20    continue # Wait for writer to finish
21  mutex_unlock(&rwlock.wlock) # Release the mutex
22
23 def up_read(rwlock): # Release read lock
24  atomic_dec(&rwlock.rwcounter, RCNT) # Decrease reader count
25
26 def down_write(rwlock): # Acquire write lock
27  # The writer tries to set the WP byte (as 1)
28  if CAS(&rwlock.rwcounter, 0, WP) == 0:
29    return # Writer fastpath
30
31  # Switch to the ephemeral stack and acquire lock in slowpath.
32  cst = this_cpu_comb_struct() # Get the per-CPU comb struct
33  switch_stack(rwlock, cst)
34  return
35 def lock_slowpath(lock, cst): # Write lock slowpath
36  ...
37  - # Waiter → queue's head; get the TAS lock
38  - while True: # Wait for the glock to be unlocked
39  -   while lock.glock != G_UNLOCK:
40  -     continue
41  -   if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
42  -     break # Got the TAS lock
43
44  + # Replace spinning on glock with rwcounter
45  + mutex_lock(&lock.wlock) # Acquire mutex
46  + if CAS(&lock.rwcounter, 0, WP) == 0:
47  +   goto unlock # Success if no readers are present.
48
49  + atomic_inc(&lock.rwcounter, WW) # Indicate writer waiting
50  + while True: # Spin until all readers finish CS
51  +   if CAS(&lock.rwcounter, WW, WP) == WW:
52  +     break
53  +   unlock:
54  +   mutex_unlock(&lock.wlock) # Release mutex
55  +   # MCS unlock phase
56  +   ...
57
58  - # Now, qnext is the combiner, indicated by glock word
59  - lock.glock = G_LOCKED_COMBINER
60  + # Now, qnext is the combiner, indicated by rwcounter word
61  + lock.rwcounter = WCOMBINER
62
63  - # Combiner phase is over, now combiner will run its CS
64  - lock.glock = G_LOCKED # Reset TAS lock to normal lock
65  + lock.rwcounter = WP
66
67 def up_writer(rwlock): # Release write lock
68  + if rwlock.rwcounter == WP:
69    rwlock.rwcounter = 0
70  ...

```

Listing 2: Pseudo-code for  $\text{TCLock}^{RW}$ .

byte (`WW`) indicating a writer waiting to acquire the lock. 2) A writer queue (`tail`) for combining and parking waiting writers. This queue is similar to our  $\text{TCLock}^B$ 's queue. 3) A

mutex, called `wlock`, that synchronizes the phase between readers and the head of the writers queue. Hence, `wlock` handles the parking of readers and the head of the write queue. We use the `ShflLockB` algorithm [52]—a traditional NUMA-aware queue-based mutex—for `wlock` than `TCLockB` because maintaining a centralized count of readers (shared data) contradicts the design of combining that tries to localize the access to the shared data.

**Algorithm.** Listing 2 shows the necessary changes to `TCLockSP`. A reader first atomically increments `RCNT` and executes its critical section if no writer is present (lines 10–11). Otherwise, it decreases the `RCNT` (line 13), and enters the slow-path phase. The reader first acquires `wlock` (line 17), it then increments the `RCNT` (line 18) to mark that a reader phase should begin soon, and waits for existing writers to exit (line 20). Finally, it unlocks `wlock` (line 21) and executes its critical section. In the unlock phase, a reader releases the lock by atomically decreasing `RCNT` (line 24).

A writer enters the critical section if it successfully switches `WP` from 0 to 1 (line 28). Otherwise, it switches to the ephemeral stack and begins the slow path phase (line 33). This slow path follows the same protocol as `TCLockSP` except that the head of the waiting queue (line 44) acquires the `wlock` (line 45). After acquiring `wlock`, the writer tries to enter the critical section by atomically setting the value to `WP` (line 46). On failure, it sets the `WW` byte to 1 to prevent new readers from entering the critical section (line 49) and waits for other readers to leave. Once they leave, the writer atomically modifies the `rwcounter` from `WW` to `WP` (line 50–52), releases `wlock`, and starts the combining process. In the unlock phase, a writer resets the `rwcounter` to 0 if the value is `WP`.

### 3.6 Optimizations

We propose three key optimizations to minimize further the data movement between the combiner and a waiter, and the cache-line bouncing of the lock word.

#### 3.6.1 Direct stack switching: waiter → waiter

In the current `TCLockSP` version (§3.2), the combiner switches stack twice before executing the next waiter’s critical section. Specifically, it first switches to its own context, finds the next waiter to combine, and then switches to the next waiter’s context. To avoid the switch to the combiner’s context, we split the combining loop. In particular, after switching the stack to a waiter’s context, the combiner tries to select the next waiter to combine after the current waiter (4a), and notifies the previous waiter that its critical section execution is over (4c). The combiner then exits the lock function call and executes the critical section of the current waiter. After executing the critical section, *i.e.*, in the unlock phase of the current waiter, the combiner checks the combining loop conditions (4d). If the condition holds, the combiner directly switches to the next waiter’s context (4b).

Otherwise, it marks the end of the combining phase, switches back to its context, notifies the previous waiter, and finally executes its own critical section.

#### 3.6.2 Minimizing context switch overhead

Our combining approach suffers from saving, transferring, and restoring a thread’s contexts while executing the critical section. We leverage both compiler and hardware techniques to minimize extra latency incurred inside the critical section.

**Leveraging function’s caller-callee convention.** Our basic context-switch algorithm saves, transfers, and restores all CPU-specific registers. We minimize this overhead by leveraging the function calling convention. Specifically, we explicitly make the slow-path of lock acquisition as a function to prevent compiler inlining. This has two benefits: First, this phase is triggered only in the case of contention, as in the uncontended case, the thread acquires the TAS lock. This approach is similar to the Linux spinlock implementation [41]. Second, we leverage the function calling convention. In particular, we save, transfer, and restore only the callee-saved registers, while the compiler saves and restores the necessary caller-saved registers [3]. The compiler, using its register liveness information, knows exactly which caller-saved registers are in use when the slow-path function is called, and spills only those registers to the stack. Moreover, the number of callee-saved registers is small [1, 8, 10]. For example, with `x86_64`, there are only six callee-saved registers compared with 16 general-purpose registers. Thus, the combiner only transfers at most one cache line to encapsulate any critical section of arbitrary length.

**Prefetching thread-local data.** Accessing thread-local data within the critical section requires moving the waiter’s specific code and data residing on its CPU to the combiner’s CPU. Unfortunately, this movement extends the length of the critical section. We find that most of the local data resides on the stack due to aggressive compiler optimizations that a thread either accesses or modifies in the critical section. Thus, we prefetch contiguous cache lines from the top of the stack<sup>2</sup> to minimize the critical section latency. The combiner issues the prefetch requests before executing the current waiter’s critical section. Traditional lock designs cannot adopt this approach because, unlike the combining approach, the lock holder already has access to its local data. Meanwhile, shared data cache lines are always going to move to the CPU holding the lock, and their movement is only possible at the end of the critical section.

#### 3.6.3 NUMA awareness

Similar to `SHFLock`, `TCLock` can adopt different policies to choose the next waiter (`select_next_waiter()` in §3.2). `TCLock` currently employs a NUMA-aware policy that minimizes cache-line bouncing among NUMA nodes. In particular, the combiner only executes critical sections of waiters

<sup>2</sup>We prefetch data in the write mode using the `prefetchw` instruction.



belonging to the same NUMA node. Upon reaching the combining limit, the combiner passes the role to a waiter residing on another NUMA node. TLock adopts the dynamic queue-splitting approach from the CNA algorithm. Specifically, TLock maintains a combiner-local waiting queue for remote NUMA node waiters and uses the primary queue for local NUMA node waiters. Initially, all waiters join the primary queue (same as before), and the selection of the next waiter happens as follows: The combiner tries to find the next waiter from the same socket. If it succeeds, it executes that waiter's critical section; otherwise, it adds the remote waiter's node to its local queue. While passing the role, the combiner first enqueues the local queue waiters at the beginning of the primary queue and then passes the combining role to the primary queue head.

## 4 TLocks with Real-World Applications

Although TLocks offer a compelling case to minimize overall shared data movement, applying them to real-world applications introduces two major challenges. The first challenge involves fine-grained locking, which requires support for multi-level locking [4, 20, 28] and out-of-order unlocking [12, 13]. The second challenge stems within the OS kernel, such as Linux, in which locks can be acquired within special execution contexts, which guarantees stability about per-CPU variables while executing within these contexts. We now discuss our approaches to overcoming these challenges in the context of the Linux kernel.

### 4.1 Multi-level Locking

Multi-level locking leads to two notable usage patterns that require additional effort to design and implement correctly in the context of combining. First, a combiner thread can be a waiter while executing a nested lock (henceforth called *waiter-combiner*). Second, locks can be released in arbitrary order leading to out-of-order unlocking. Although out-of-order unlocking does not affect traditional locks, TLock requires extra care in handling such cases, as it can lead to data corruption as well as deadlocks. We now present our approach to supporting these usage patterns.

**Nested combining.** For handling nested combining, TLock adopts the same interrupt processing mechanism by OSes. Interrupt handlers, before processing the interrupt, push the current thread state on the stack. When the interrupt handler finishes, it restores the thread's state from the stack. This allows for handling nested interrupts without affecting the execution of the interrupted thread. There are three cases that occur when TLocks interplay in the context of nested locking: First is the case of nested locks when both locks are in their combining phase. The second one is when the outer level lock is a combiner and the inner one is the fast-path TAS lock. Finally, the third case is the opposite of the second scenario.

Listing 3 shows the changes required to implement the first case, which works similar to the interrupt processing

```

1 + G_UNLOCKED_000 = 4
2 # Extra state to handle out-of-order unlocking
3
4 def switch_stack(lock, cst):
5     switch_to_ephemeral_stack(cst.node) # Main → ephemeral
6 - lock_slowpath(lock, cst)
7 + ret_val = lock_slowpath(lock, cst)
8 + if ret_val == G_UNLOCKED_000: # Only for nested combiner
9 +     switch_to_combiner_previous_stack_frame()
10 + else:
11 +     switch_from_ephemeral_stack(cst.node) # Ephemeral→main
12
13 def lock_slowpath(lock, cst):
14     ...
15     if qprev is not None: # Waiters are in the queue
16         ...
17         if qnode.request == PRCSO : # Waiter rqst is processed
18 +         if lock.glock == G_UNLOCKED_000: # 000 unlock
19 +             return G_UNLOCKED_000
20             return 0 # Combiner executed my CS
21         ...
22
23     # For handling nested combining
24 + save_on_curr_stack_frame(lock.glock, cst.qcurr, qnode.rsp)
25
26     lock.glock = G_LOCKED_COMBINER
27 + j = find_first_empty_index(cst.lock_addr)
28 + cst.lock_addr[j] = lock # Record the current lock address
29     # Combiner loop ...
30     # Combiner phase is over, now combiner will run its CS
31 + cst.lock_addr[j] = None # Remove the current lock address
32
33     # For handling nested combining
34 + restore_from_curr_stack_frame(lock.glock,
35 +                               cst.qcurr, qnode.rsp)
36     notify_next_queue_head(qnext) # Next waiter is the combiner
37
38 def spin_unlock(lock):
39     if lock.glock == G_LOCKED:
40         lock.glock = G_UNLOCK # Only true for no combining phase
41         return
42 + max_idx = find_last_not_empty_index(cst.lock_addr)
43 + my_idx = find_my_lock_index(cst.lock_addr, lock)
44 + if my_idx < max_idx : # Acquire order != release order
45 +     lock.glock = G_UNLOCKED_000
46 +     return
47     # Jump back to combiner
48     cst = this_cpu_komb_struct()
49     # Waiter's stack → combiner's ephemeral stack
50     switch_stack_from_waiter_to_combiner(cst.qcurr, cst)
51     return

```

Listing 3: Out-of-order (OOO) unlocking protocol.

approach. Specifically, a combiner may acquire a nested lock inside the critical section. Before acquiring that lock, the combiner pushes its state onto its ephemeral stack (line 24), which it restores after releasing that nested lock (line 35). This allows TLocks to handle arbitrary levels of nesting without violating application correctness.

TLocks also supports the last two scenarios, in which one of the locks is in the combining phase. We do not require any additional support for these cases because each lock is independent and the underlying lock mechanism doesn't interact with each other, which is exactly the same in the traditional lock design. Specifically, every lock has its own lock word and its underlying lock mechanism only interacts with its own lock word. Thus, acquiring the lock in the fast-path (TAS lock), does not interact with the lock which is held by the combiner thread.

**Out-of-order (OOO) unlocking.** The algorithms discussed thus far for TLocks incorrectly handle OOO unlocking, leading to wrong program execution. We illustrate this through an example: Suppose multiple threads acquire  $L_A$  and then  $L_B$ , which leads to contention. As a result, the combining phase becomes active, and  $C_A$  and  $C_B$  hold locks  $L_A$  and  $L_B$ , respectively.  $C_A$  becomes a waiter-combiner when it tries to acquire lock  $L_B$ . Now, if  $L_A$  is released before  $L_B$ —unlocked OOO—the combiner  $C_B$  will return to its own combining loop, as the unlock function does not track of the order of unlocked locks. Therefore, the combiner  $C_B$  breaks application semantics by starting to execute the next waiter’s critical section, while the lock  $L_B$  which it holds is not unlocked yet.

To handle OOO unlocking, we rely on a simple insight: we can release a lock at a later point in time without affecting the correctness of the application. Specifically, we do not release  $L_A$ ; we only release it once  $L_B$  has been released. This approach is similar to the handling of nested transactions, in which we effectively flatten the out-of-order lock hierarchy and release all the locks at the same time.

Listing 3 shows the changes required to TLocks for handling OOO unlocking. We make three specific changes in the lock and unlock function of TLocks:

- To identify OOO unlocks, we maintain a per-thread `lock_addr` array to record the acquisition order of locks. Before starting the combining loop, a combiner stores its lock’s address in the `lock_addr` array (line 28), and removes it once the combiner loop finishes (line 31).
- In the unlock function, the combiner checks if the lock that is being unlocked is the last entry in the `lock_addr` array. This is because the last entry in the `lock_addr` array is the lock holding which the current combiner is executing the critical section. We have two cases now: The first one is the non-OOO case (line 47): The combiner follows the original algorithm and returns to the combiner’s ephemeral stack (line 50) and continues with its combiner loop. While the second one is an OOO case (line 44): We simply mark the lock as *OOO-unlocked* (line 45) and let the current combiner continue executing until the unlock function for its lock is called. The waiter-combiner for the OOO-unlocked lock waits for the notification from the current combiner which doesn’t happen until the current combiner reaches its combiner loop. Therefore, delayed notification effectively flattens the lock hierarchy for out-of-order unlocked locks as the waiter-combiner cannot progress until it gets the notification.
- After receiving the notification, the waiter-combiner checks if its lock is unlocked OOO (line 18) and if true, it return `G_UNLOCKED_OOO` (line 19). The combiner switches to its previous state (line 9) which was saved when the nested lock was called (line 5). The combiner returns to its combiner loop (line 29), notifies the current waiter and continues combining the next waiter.

The waiter for the outermost lock will get the control back once the outermost lock and all the nested locks are released. The waiter then starts executing its non-critical section.

## 4.2 Special Execution Contexts and Per-CPU Variables

Delegation via transparent combining breaks assumptions of Linux kernel code about the stability of access to per-CPU variables under special execution contexts. This includes interrupt handlers, non-preemptible contexts, non-migratable contexts, etc. This raises a critical question for our design: How do we enable delegation-style locking transparently in the kernel without compromising on correctness?

A potential solution involves the combiner accessing the per-CPU variables of the waiter’s CPU while executing the critical section. For example, on x86, we can save and restore the `gs` registers that allows access to per-CPU variables of the waiter’s CPU [14]. Unfortunately, this approach leads to data races when waiters are busy-waiting, as interrupts on the waiter’s CPU may still access per-CPU variables. Moreover, this approach further leads to additional overhead of accessing per-CPU data of a remote CPU. Besides that, it requires annotating parts of the kernel code that access per-CPU variables for functional correctness, such as scheduler [22], RCU [18], and many more during the combining phase. As a result, these challenges make it very difficult to enable transparent combining in special execution contexts within the kernel.

We adopt a more conservative approach of disabling combining for such execution contexts and falling back to default kernel locking (currently `qspinlock` [41]). We leverage the property that any part of a critical section requiring stable access to per-CPU variables ensures appropriate protection against CPU migration for that region of code. For example, the Linux kernel’s `spinlock_t` APIs do not guarantee stable access to per-CPU variables, as they do not disable preemption for the critical section. This is because the `spinlock_t` type is transparently replaced with a mutex on real time kernels [19]. Hence, the scheduler is allowed to preempt threads and migrate them to a different CPU when they are holding such a lock. To ensure that preemption is disabled within the critical section regardless of the kernel configuration, developers use specific `raw_spinlock_t` APIs [17].

When invoking the TLock APIs, we only enable combining for threads that can migrate from one CPU to another. Otherwise, we disable combining and fallback to the existing traditional lock. We identify these code regions by leveraging well-defined APIs of the Linux kernel. In particular, we enable combining for the following cases: 1) the kernel thread executes in the *task* context [27]; 2) it does not disable *migration* or *preemption* [9, 16], and 3) it does not execute in a context where *HardIRQs* and *SoftIRQs* are disabled [5, 24]. It is safe to execute traditional and combining queue-based lock because mechanism for both types of locks are inde-

pendent and only one of them will be active for a particular instance of lock at any given point.

## 5 Implementation

We implement TLocks in the Linux kernel v5.14.19 and replace all *spinlock*, *mutex*, and *rwsem*. We add 1349, 955, and 1652 LoC for spinlock, mutex, and readers-writer lock (rwsem), respectively. For userspace applications, we use LiTL [50] library. It uses the LD\_PRELOAD mechanism to interpose different POSIX locks used by userspace applications.

We implement TLock for the x86 architecture, but it is easily extensible to other architectures as well. x86-64 has six callee-saved registers: *rbx*, *rbp*, and *r12-r15*. We push these registers on the stack along with the stack pointer on the waiter’s qnode. When the combiner switches to the waiter’s main stack, it uses the stored stack pointer and pops the callee-saved registers from the stack. We mark the stack-switch function as *nonline* and *noipa* to prevent any compiler optimizations and function inlining. Our code is publicly available here: <https://github.com/rs3lab/TLocks>.

## 6 Evaluation

We evaluate TLocks by answering the following questions:

- Q1. How does the kernel-based TLock implementation impact micro-benchmarks (§6.1) and real applications (§6.2)?
- Q2. How does each design decision affect TLocks’ performance (§6.3)?
- Q3. How does the userspace TLock implementation impact an application’s performance (§6.4)?

**Evaluation setup.** We use micro-benchmarks that mostly stress a lock and application benchmarks that stresses various kernel subsystems. In addition, we use a hash-table nano-benchmark [69] to show the effectiveness of TLock design decisions. We evaluate on an 8-socket, 224-core Intel machine with hyper-threading disabled. We use *tmpfs* in all experiments to minimize the file system overhead. We evaluate three traditional locks within the Linux kernel: Linux’s stock locks, CNA, and SHFLLOCK. CNA replaces the stock *qspinlock*, while we replace all locks with SHFLLOCK.

### 6.1 TLock Performance Comparison

We evaluate TLocks using a set of micro-benchmarks [31, 62]. Each micro-benchmark instantiates a set of threads and pins them to cores. These threads mostly contend on a single lock (sometimes two) while performing specific tasks (Table 1) for 30 seconds.

**Spinning TLock.** Figure 3 ((a) and (b)) show that TLock<sup>SP</sup> outperforms the Linux version (Stock) by 3.7× and 4.4× on MRDM and lock1, respectively. TLock performs similarly to Stock from two to eight cores for two reasons. First, the stack switching adds an average of 47 ns latency. Second, the combiner is unable to perform effectively at such a low core count. As a result, TLock does not reach its potential. On the other hand, the benefit of

Lock type	Workload	Lock: Usage
Spinlock	MRDM [62] lock1 [31]	<b>rename seqlock:</b> Rename files within a directory <b>files_struct.file_lock:</b> fd allocation / fcntl
Blocking	MWRM [62]	<b>sb-&gt;s_vfs_rename_mutex:</b> Rename a file across directory <b>dentry-&gt;d_lock:</b> Directory lock
RW Blocking	mmap1 [31]	<b>mm_struct-&gt;mmap_lock:</b> Memory map file within a directory

Table 1: Lock usage in various micro-benchmarks [31, 62].

TLock<sup>SP</sup> is evident after eight cores where the gains from localizing shared data cache lines outweighs the overhead of stack-switch. TLock<sup>SP</sup> combiner on average batches 950 waiter’s request. Thus, even within a socket (up to 28 cores), TLock<sup>SP</sup> maintains consistent throughput.

Compared to NUMA-aware locks, TLock<sup>SP</sup> outperforms SHFLLOCK and CNA by 2-3× across sockets. The combining-based NUMA-aware policy of TLock<sup>SP</sup> minimizes the cache-line bouncing of both the lock word and the shared data. On average, 190K combiners execute during a 30-second run where every TLock<sup>SP</sup> combiner batches ~980 waiter’s request before passing the lock to different NUMA socket. In essence, every combiner is reducing extra coherence traffic for accessing shared data within the waiter’s critical section, which would be generated in a traditional lock design.

**Blocking TLock.** We compare TLock<sup>B</sup> with Linux *mutex* and SHFLLOCK. Figure 3 (c) shows that TLock<sup>B</sup> is 1.8× faster than both *mutex* and the blocking version of SHFLLOCK. Both *Stock* and SHFLLOCK suffer from shared data movement at a high core count. In addition, SHFLLOCK’s performance degrades similarly to that of *Stock* due to its lock stealing, which renders its NUMA-policy ineffective. Whereas, TLock<sup>B</sup> retains performance because it reduces cache-line bouncing for both the lock word and shared data.

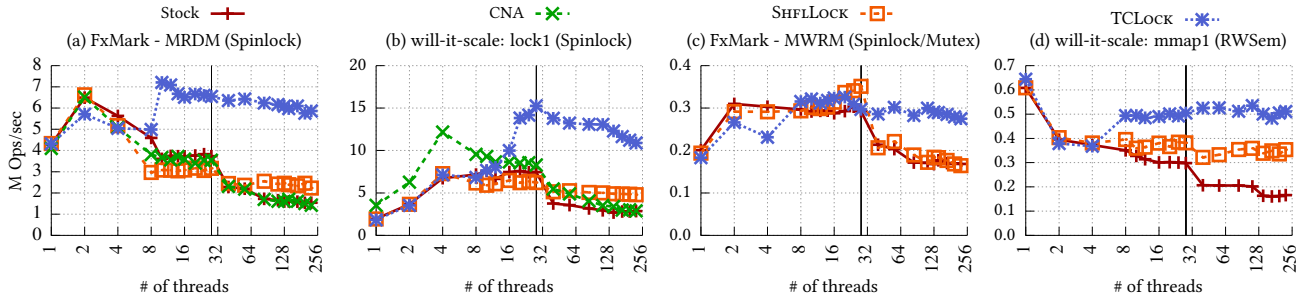
**Readers-Writer Blocking TLock.** Figure 3 (d) shows the impact of TLock when stressing the writer side of *rwsem*. We use the *mmap1* benchmark [31], which populates and deletes VMAs within an address space. TLock maintains the best throughput irrespective of contention after eight cores. Within a socket, TLock<sup>RW</sup> outperforms *Stock* by 1.7×, as a combiner combines ~1000 waiter’s request, thereby minimizing cache-line movement of shared data cache lines. Moreover, across the socket, TLock<sup>RW</sup> combiner batches similar number of waiter’s request resulting in 3.1× and 1.5× better throughput than *Stock* and SHFLLOCK.

### 6.2 Application-level Benchmarks

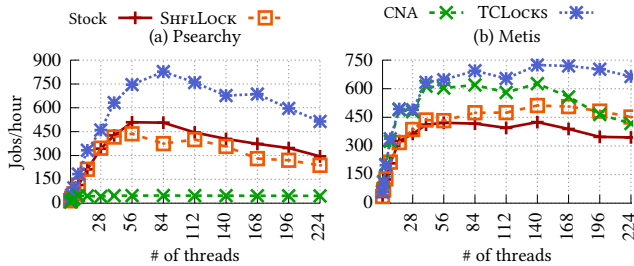
We evaluate two applications that extensively stress various subsystems of the Linux kernel. Figure 4 reports applications’ throughput. The kernel subsystem uses a mix of blocking locks and spinlocks, which are present in several data structures such as inodes, task structures, and memory mappings.

**Psearchy** is a parallel version of *searchy* that does text indexing. It is *mmap* intensive, which stresses the memory subsystem with multiple userspace threads. It does around 96,000





**Figure 3:** Impact of spinlock, blocking locks and read-write semaphore on the scalability of micro-benchmarks [31, 62].



**Figure 4:** Impact of kernel locks on application scalability.

small and large mmap/munmap operations from 96,000 files with multiple threads. It stresses the writer side of the `rwsem` in the memory subsystem and inode allocation in the file system layer. Figure 4 (a) shows that TLOCK outperforms existing locks up to  $2.2\times$ . Because of its effective combining strategy, TLOCK<sup>RW</sup> is able to localize access to shared data. We find that SHFLLOCK and Stock have similar performance as they inefficiently use up hardware bandwidth. Moreover, we observe that psearchy’s performance drops with increasing core count, which happens due to the contention in the file stream glibc library.

**Metis** is an in-memory map-reduce framework, representing a page-fault-intensive workload that stresses the readers’ side of the `mmap_sem` (`rwsem`) in the Linux kernel. Figure 4 (b) shows that TLOCK outperforms both SHFLLOCK and Stock by  $1.3\times$ . The reason is due to the phase-based design of TLOCK<sup>RW</sup>, which improves the performance by batching the writers in one phase, meanwhile executing readers in parallel in the next phase. Across sockets, it improves performance compared to SHFLLOCK and Stock by  $1.7\times$  and  $1.4\times$  at 140 cores, respectively.

### 6.3 Nano benchmark: RCUHT

We now do an in-depth analysis of TLOCKS using a hash-table benchmark in the kernel [69]. A global lock guards the hash table. For TLOCK<sup>SP</sup> and TLOCK<sup>B</sup>, we generate 100% writes, whereas for TLOCK<sup>RW</sup> (readers-writer blocking lock), we generate 1% and 20% writes on the hash table. Figure 5 presents the results and the factor analysis of TLOCKS.

**Spinning TLOCK.** Figure 5 (a) and (b) shows the throughput and 99.99% latency of spinlocks, respectively. (a) We find that TLOCK<sup>SP</sup> maintains similar performance within and across sockets because of the effective combining pol-

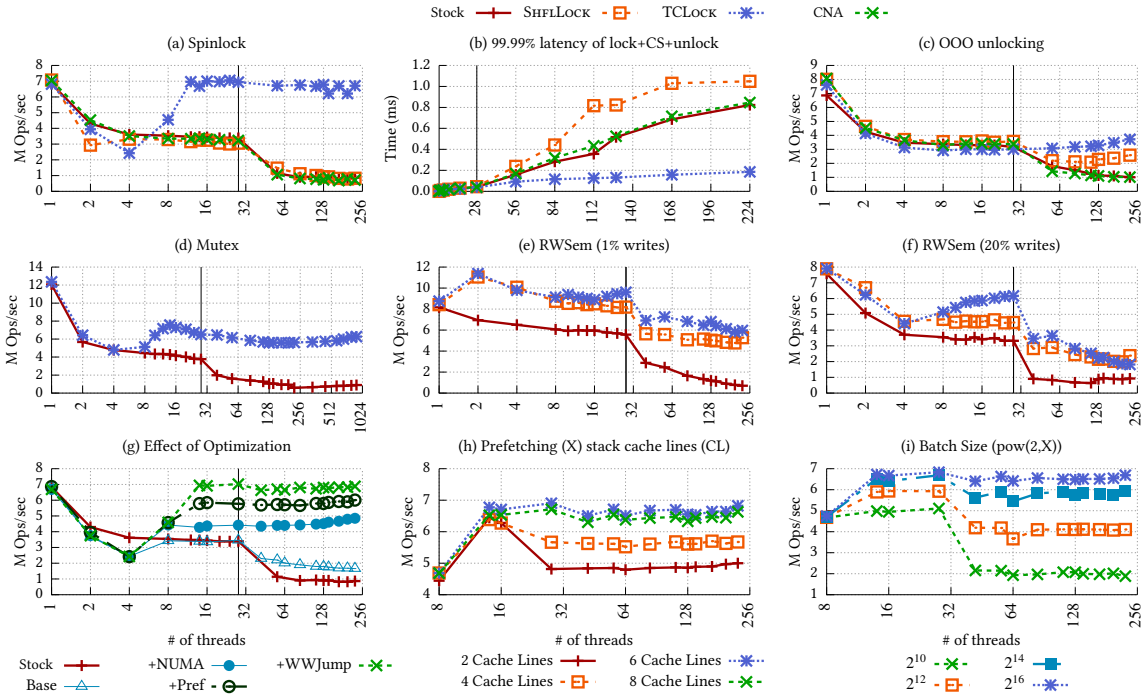
icy. In particular, the combining batches up to 50,000 waiter requests, thereby localizing the requests for that many invocations. In addition, the average and 99%ile latency of the critical section is 188 ns and 474 ns at 28 cores, respectively, whereas both stock and SHFLLOCK have up to  $2.5\times$  and  $2.1\times$  higher average and 99%ile latency, respectively.

In the case of NUMA, TLOCK<sup>SP</sup> outperforms SHFLLOCK and Stock by up to  $9.4\times$ . The improvement occurs because of minimizing cache-line bouncing and Localizing shared data, which reduces the time spent in critical section. For example, at 168 cores, the average latency of TLOCK is 213ns, which is similar to average latency at 28 cores. Whereas SHFLLOCK and Stock have  $10.5\times$  and  $11\times$  higher latency, respectively. The 99%ile latency increases to 1516 ns for TLOCK. This happens because of NUMA-aware moving of the shared data, which increases the 99.9% latency of TLOCK. However, this latency is still  $3\times$  lower than that of SHFLLOCK.

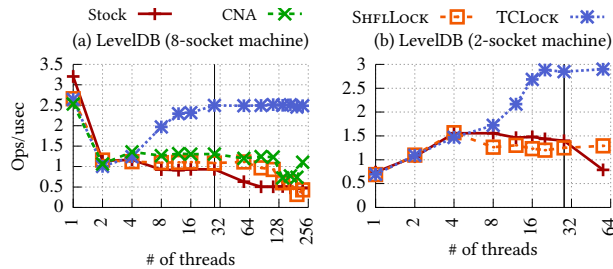
Figure 5 (b) shows the combined latency for the lock function, critical section execution, and the unlock function. TLOCK<sup>SP</sup>, even with batching 50,000 waiters, has up to  $5.6\times$  and  $4.4\times$  lower latency compared to SHFLLOCK and Stock, respectively. This is because of lower critical section latency, which reduces the overall latency of the whole system as the critical sections are executed sequentially.

**Nested Locking and OOO unlocking.** We evaluate the impact of our OOO unlocking with a hash-table nano-benchmark that acquires nested lock and can release locks in an OOO manner. Specifically, every bucket has a lock and nested locks are acquired when moving an entry from one bucket to another. Figure 5 (c) shows that, within a socket, TLOCK<sup>SP</sup> performs similar to other locks. At 28 cores, TLOCK<sup>SP</sup> is 5% slower than Stock. This is because the overhead of saving/restoring the combiner state with nested lock along with delaying the unlock degrades TLOCK<sup>SP</sup> performance within a socket. Across socket, TLOCK<sup>SP</sup> outperforms Stock by up to  $3.7\times$ . The performance gains with localization of shared cache lines outweighs the overhead of TLOCK<sup>SP</sup> implementation of nested locking.

**Blocking TLOCK.** Figure 5 (d) shows the throughput with blocking locks. TLOCK<sup>B</sup> outperforms stock in both under-subscribed and over-subscribed scenarios. With the help of its efficient spin-then-park strategy, TLOCK<sup>B</sup> outperforms Stock by up to  $9.5\times$  in under-subscribed scenarios. More-



**Figure 5:** (a – f) Impact of spinning, blocking and read-write lock on the hash-table nano benchmark with an eight-socket Intel machine. (b) Latency of executing lock function+CS+unlock function with different spinlocks. (c) Performance with nested locking and OOO unlocking. (g) Impact of different optimization introduced in  $\text{TCLock}^{SP}$ . On top of baseline, we add NUMA-awareness, stack prefetching, and waiter-to-waiter jump. (h – i) Impact of prefetching and batch size on  $\text{TCLock}^{SP}$ 's performance.



**Figure 6:** Impact of locks on userspace applications.

over,  $\text{TCLock}^B$  maintains the same performance even after crossing the socket boundary. Although latency to wake up a waiter on a remote socket costs more than that of the local socket,  $\text{TCLock}^B$ 's usage of NUMA-aware design amortizes the overhead of waking up waiters from other socket.

**Reader-writer  $\text{TCLock}$ .** Figure 5 ((e) and (f)) show that the  $\text{TCLock}^{RW}$  has higher throughput than the stock version by 6.8 $\times$  and 2.2 $\times$  for 1% and 20% writes, respectively.  $\text{SHFLock}$  and  $\text{TCLock}^{RW}$  use similar design for readers. Because of using the phase-based design,  $\text{TCLock}^{RW}$  is able to improve performance by up to 1.28 $\times$  and 1.37 $\times$  at 1% and 20% writes, respectively. We further observe that combining is not effective with centralized readers counting, as the readers counter cache line is always moving across cores.

**$\text{TCLocks}$  optimizations.** Figure 5 (g) shows the effect of

different optimizations used by  $\text{TCLock}^{SP}$ .  $\text{TCLock}^{SP}$  without any optimizations outperforms Stock by 2.2 $\times$  because it localizes the shared data access. The overhead of stack switch is apparent at lower core count because jumping to a waiter's critical section requires access to the stack which needs to be fetched from a waiter's core. On adding NUMA-awareness to the current design, we improve the performance by 2.6 $\times$ , as we now prevent moving the waiter's stack cache line across sockets. It also helps within a socket because checking the socket ID of the next waiter's node fetches the next waiter's node in the combiner's cache. As a result, this simple check reduces the time spent in the combiner loop.

In addition, our stack prefetching approach, on top of NUMA-awareness policy, further improves performance by 1.3 $\times$ , as it reduces the time spent in starting the execution of critical section. Finally, our waiter  $\rightarrow$  waiter jump (WWJump) further improves the throughput by 1.2 $\times$  as it reduces the overhead of stack switch ( $\sim 50$  ns) from two switches to one. Overall, our optimizations reduce the overhead of stack switching and improve performance compared to the baseline by 4 $\times$ .

**$\text{TCLocks}$  sensitivity.** Figure 5 ((h)–(i)) shows the impact of changing the number of prefetched cache lines and the number of waiter's combined. Figure 5 (h) shows that prefetching up to six cache lines provides the best performance for this

benchmark. It depends entirely on what is accessed inside the critical section. We can write a compiler pass to tune this parameter, as the compiler has the information on what is accessed within the critical section. Figure 5 (i) shows the impact of batching. Higher batch count improves throughput at the expense of short-term fairness, but TLocks maintain long-term fairness. Batching is also able to reduce the latency for all requests, if it can reduce the time spent per request as shown in Figure 5(e).

#### 6.4 Performance With Userspace TLock

We evaluate TLocks on the LevelDB benchmark [49]. We integrate both TLock, CNA and SHFLock into LiTL [50] for evaluation. LevelDB is an open-source key-value store [48]. We use the readrandom benchmark with 1M key-value pairs, that contends on the global database lock. Figure 6 (a) shows the performance with spinlocks on an 8-socket machine. Within a socket, TLock<sup>SP</sup> improves throughput compared to other locks by  $1.9\times$ – $2.6\times$ . Localizing shared data movement helps to achieve better performance than traditional locks. Across sockets, NUMA-awareness coupled with minimal shared data movement helps TLock<sup>SP</sup> outperform other locks by up to  $5.2\times$ . Figure 6 (b) shows the performance on a 2-socket machine. TLock<sup>SP</sup> performs similar to the 8-socket machine and improves throughput compared to other locks by  $2.1\times$ – $3.6\times$ .

### 7 Discussion and Limitations

TLocks implement transparent delegation, which enables developers to use delegation-style locking without rewriting the application. However, TLocks have limitations both in terms of algorithm design and kernel implementation. We discuss them below.

**Overhead at two–four cores.** We observe overhead with TLocks when very few threads (two–four) contend for a lock. Contending threads execute slowpath after stack-switching, but combining is only enabled when more than two waiters are present in the queue. Waiters pay the cost of two stack-switching but their critical section is not executed by the combiner. This can be solved by disabling combining when we have less than four threads in the queue. The challenge lies in efficiently identifying the size of the queue without using extra memory or traversing the queue.

**Resource accounting.** The kernel requires accurate accounting of resources like CPU usage, allocated memory, etc. Kernel subsystems, such as the scheduler or cgroup, are guided by the accounting of resources used by a particular thread. Delegation-based techniques can break this accurate accounting for resources used within the critical section. Thus, TLocks complicate resource accounting. Even though a combiner thread executes the critical section on behalf of other waiter threads, resources like CPU time or allocated memory in the critical section need to be accounted to the waiter thread, for maintaining broader kernel

semantics. We leave this extension as future work.

**TLock vs ‘current’.** Apart from per-CPU variables, Linux also uses a macro named *current*, which resolves to a per-CPU pointer variable to the currently executing thread’s task structure. This pointer is used to access the task structure for multiple purposes, including but not limited to resource accounting with cgroups [21], permission checks using credentials [15], thread scheduling [26], etc. While executing a waiter’s critical section on the combiner’s CPU, if this pointer is not switched to the waiter’s task structure, then it could lead to subtle bugs. For example, if a combiner thread has higher privileges than the waiter thread, and the permission checks are done within the critical section, it may lead to privilege escalation bugs, since the combiner thread’s credentials will be inspected.

One possible solution is to modify *current* macro’s implementation to resolve to the waiter’s task structure while executing waiter’s critical section on combiner CPU. Unfortunately, this will also lead to bugs. For example, if a thread sleeps within its critical section, the scheduler code uses the *current* macro to put the running task to sleep. When combining a waiter’s critical section, we want the combiner thread to sleep. However, if we switch the *current* macro to use the waiter’s task structure, it will lead to confusion within the scheduler as the waiter task is already seen to be running on another CPU.

We currently keep the *current* macro unchanged, and suggest judicious use of the TLock APIs in cases where a different thread identity within the critical section may lead to unexpected behavior.

### 8 Conclusion

Delegation based techniques are known to offer better scalability and provide better performance for highly contended scenarios, but prior work requires application changes to enable delegation. In this paper, we propose a new technique called *transparent delegation* that makes delegation-style locking practical. We design the first-ever transparent delegation based locks, called TLocks, for both userspace applications and the Linux kernel. This is achieved by lightweight context switching and using ephemeral stacks to maintain consistency. Using transparent delegation, we design spinning, blocking and phase-based readers-writer locks. We replace all the locks in the Linux kernel with TLocks, and discuss the technical challenges involved. Our evaluation shows that TLocks provide better performance and scalability compared to traditional lock design.

### 9 Acknowledgment

We thank Changwoo Min for his comments on the initial draft. We also thank Dave Dice, Alex Kogan, the anonymous reviewers, and our shepherd, Geoffrey M. Voelker, for their helpful feedback. This work is supported by the SNSF project grant 212884.



## References

- [1] List of callee-saved registers. <https://developer.arm.com/documentation/102374/0100/Procedure-Call-Standard>, [Accessed on 22/04/2023].
- [2] List of arm registers. <https://developer.arm.com/documentation/dui0473/m/overview-of-the-arm-architecture/arm-registers>, [Accessed on 22/04/2023].
- [3] Gcc calling convention. <https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html>. [Accessed on 22/04/2023].
- [4] Lock ordering for file mmap. <https://elixir.bootlin.com/linux/v6.1/source/mm/filemap.c#L72>. [Accessed on 22/04/2023].
- [5] Locking Between Hard IRQ and Softirqs/Tasklets: Unreliable Guide To Locking – The Linux Kernel documentation. <https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html#locking-between-hard-irq-and-softirqs-tasklets>, [Accessed on 30/04/2023].
- [6] Hardirq. <https://www.kernel.org/doc/htmldocs/kernel-locking/hardirq-context.html>, [Accessed on 22/04/2023].
- [7] Hardware interrupts (hard irq). <https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-hardirqs.html>, [Accessed on 22/04/2023].
- [8] List of x86-64 registers. [https://wiki.cdot.senecacollege.ca/wiki/X86\\_64\\_Register\\_and\\_Instruction\\_Quick\\_Start](https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start). [Accessed on 22/04/2023].
- [9] [PATCH 7/9] sched: Add migrate\_disable(). <https://lwn.net/ml/linux-kernel/20200921163845.769861942@infradead.org/>. [Accessed on 30/04/2023].
- [10] List of mips registers. [https://en.wikibooks.org/wiki/MIPS\\_Assembly/Register\\_File](https://en.wikibooks.org/wiki/MIPS_Assembly/Register_File). [Accessed on 22/04/2023].
- [11] Non-maskable interrupt. [https://en.wikipedia.org/wiki/Non-maskable\\_interrupt](https://en.wikipedia.org/wiki/Non-maskable_interrupt). [Accessed on 22/04/2023].
- [12] Dentry cache spinlock unlocked out-of-order. <https://elixir.bootlin.com/linux/v6.0/source/fs/dcache.c#L3022>.
- [13] Pipe mutex unlocked out-of-order. <https://elixir.bootlin.com/linux/v6.0/source/fs/splice.c#L1552>.
- [14] Per-cpu variables. [https://docs.kernel.org/core-api/this\\_cpu\\_ops.html#inner-working-of-this-cpu-operations](https://docs.kernel.org/core-api/this_cpu_ops.html#inner-working-of-this-cpu-operations), [Accessed on 22/04/2023].
- [15] Credentials in Linux. <https://www.kernel.org/doc/Documentation/security/credentials.txt>, [Accessed on 30/04/2023].
- [16] Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe. <https://www.kernel.org/doc/Documentation/preempt-locking.txt>. [Accessed on 30/04/2023].
- [17] raw\_spinlock\_t: Lock types and their rules – The Linux Kernel documentation. <https://docs.kernel.org/locking/locktypes.html#raw-spinlock-t>. [Accessed on 30/04/2023].
- [18] What is RCU? – “Read, Copy, Update” – The Linux Kernel documentation. <https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html>. [Accessed on 30/04/2023].
- [19] spinlock\_t and PREEMPT-RT: Lock types and their rules – The Linux Kernel documentation. <https://docs.kernel.org/locking/locktypes.html#spinlock-t-and-preempt-rt>. [Accessed on 30/04/2023].
- [20] Lock ordering for directory rename. <https://docs.kernel.org/filesystems/directory-locking.html>. [Accessed on 22/04/2023].
- [21] Control group v2. <https://www.kernel.org/doc/Documentation/cgroup-v2.txt>. [Accessed on 30/04/2023].
- [22] CFS Scheduler – The Linux Kernel documentation. <https://www.kernel.org/doc/html/next/scheduler/sched-design-CFS.html#few-implementation-details>. [Accessed on 30/04/2023].
- [23] Softirq. <https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html>, [Accessed on 22/04/2023].
- [24] Locking Between User Context and Softirqs: Unreliable Guide To Locking – The Linux Kernel documentation. <https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html#locking-between-user-context-and-softirqs>, [Accessed on 30/04/2023].
- [25] Task context. <https://www.kernel.org/doc/htmldocs/kernel-hacking/basic-players.html#basics-usercontext>. [Accessed on 22/04/2023].
- [26] CFS Scheduler – The Linux Kernel documentation. <https://www.kernel.org/doc/html/next/scheduler/sched-design-CFS.html>. [Accessed on 30/04/2023].
- [27] User Context: Unreliable Guide To Hacking The Linux Kernel – The Linux Kernel documentation. <https://www.kernel.org/doc/html/v4.16/kernel-hacking/hacking.html#user-context>. [Accessed on 30/04/2023].
- [28] Lock ordering in memory management subsystem. <https://elixir.bootlin.com/linux/v6.1/source/mm/rmap.c#L20>. [Accessed on 22/04/2023].
- [29] Windows fibers. <https://learn.microsoft.com/en-us/windows/win32/procthread/fibers>.
- [30] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.
- [31] A. Blanchard. will-it-scale. <https://github.com/antonblanchard/will-it-scale>. [Accessed on 22/04/2023].
- [32] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.
- [33] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, Oct. 2010.
- [34] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.
- [35] B. B. Brandenburg and J. H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 184–193. IEEE, 2009.
- [36] B. B. Brandenburg and J. H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. In *Real Time Systems*, pages 184–193, 2011. doi: 10.1109/ECRTS.2009.14.
- [37] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97. Springer, 2013.
- [38] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 157–166, Shenzhen, China, Feb. 2013.
- [39] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015.

- [40] J. Corbet. Big reader locks, 2010. <https://lwn.net/Articles/378911/>, [Accessed on 30/04/2023].
- [41] J. Corbet. MCS locks and qspinlocks, 2014. <https://lwn.net/Articles/590243/>, [Accessed on 30/04/2023].
- [42] T. Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, Department of Computer Science, University of Washington, 1993.
- [43] D. Dice and A. Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6281-8.
- [44] D. Dice and A. Kogan. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 173–183, 2021.
- [45] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, 2011.
- [46] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–256, New Orleans, LA, Feb. 2012.
- [47] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 257–266, New Orleans, LA, Feb. 2012.
- [48] S. Ghemawat and J. Dean. LevelDB, 2019. URL <https://github.com/google/leveldb>. [Accessed on 30/04/2023].
- [49] R. Guerraoui, H. Guiroux, R. Lachaize, V. Quéma, and V. Trigonakis. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.*, 36(1):1:1–1:149, Mar. 2019. doi: 10.1145/3301501. URL <http://doi.acm.org/10.1145/3301501>.
- [50] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 649–662, Denver, CO, June 2016.
- [51] D. Hendler, I. Ince, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.
- [52] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.
- [53] X. Leroy. The open group base specifications issue 7, 2016. <http://pubs.opengroup.org/onlinepubs/9699919799/>, [Accessed on 30/04/2023].
- [54] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, 2009.
- [55] Linux. Lock ordering, 2013. URL <https://elixir.bootlin.com/linux/latest/source/mm/filemap.c#L66>. [Accessed on 30/04/2023].
- [56] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, Jan. 2016.
- [57] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, 2006.
- [58] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.
- [59] C. D. Marlin. *Coroutines: a programming methodology, a language design and an implementation*. Number 95. Springer Science & Business Media, 1980.
- [60] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. pages 21–65, Feb. 1991.
- [61] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-writer Synchronization for Shared-memory Multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 106–113, 1991.
- [62] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.
- [63] O. Nesterov. Linux percpu-rwsem, 2012. <http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h>, [Accessed on 30/04/2023].
- [64] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, jul 1999.
- [65] S. Park, D. Zhou, Y. Qian, I. Calciu, T. Kim, and S. Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.
- [66] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1871-0.
- [67] S. Roghanchi, J. Eriksson, and N. Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 342–358, 2017.
- [68] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, Salt Lake City, UT, Feb. 2001.
- [69] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 11–11, Portland, OR, June 2011.
- [70] A. Viro. parallel lookups, 2016. <https://lwn.net/Articles/684089/>, [Accessed on 30/04/2023].
- [71] P. Zijlstra. percpu rwsem -v2, 2010. <https://lwn.net/Articles/648914/>, [Accessed on 30/04/2023].





# RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks

Shiwu Lo<sup>♡</sup>, Han-Ting Lin<sup>♡</sup>, Yao-Hung Hsieh<sup>♡</sup>, Chao-Ting Lin<sup>♡</sup>, Yu-Hsueh Fang<sup>△</sup>, Ching-Shen Lin<sup>♡</sup>,  
Ching-Chun (Jim) Huang<sup>△</sup>, Kam Yiu Lam<sup>◇</sup>, Yuan-Hao Chang<sup>°</sup>  
National Chung Cheng University<sup>♡</sup>, National Cheng Kung University<sup>△</sup>, City University of Hong Kong<sup>◇</sup>,  
Academia Sinica, Taiwan<sup>°</sup>

## Abstract

As the number of processor cores increases, the efficiency of accessing shared variables through the lock-unlock method decreases. A NUMA-aware algorithm, which only considers the transmission delay between processors, may not fully utilize the connection network of a multi-core processor. This limits the scalability of a multi-core processor due to the large amount of low- and variable-cost data sharing between cores. The problem is that the reduction in communication cost cannot compensate for the increase in the time complexity of the spinlocks, and the farthest transmission distance becomes longer with more cores.

We propose a method called Routing on Network-on-chip (RON)<sup>1</sup> to minimize the communication cost between cores by using a routing table and pre-calculating an optimized locking-unlocking order. RON delivers locks and data in a one-way circular manner among cores to (1) minimize global data movement cost and (2) achieve bounded waiting time. Microbenchmarks provide quantitative analysis, while multi-core benchmarks show performance under various workloads.

In terms of user space performance, RON improves the performance of Google LevelDB by 22.1% and 24.2% compared to ShflLock and C-BO-MCS, respectively. In the kernel space, RON is 1.8 times faster than using ShflLock for Google LevelDB. RON-plock solves the problem of oversubscription with constant space complexity and achieves 3.7 times and 18.9 times better performance than ShflLock-B and C-BO-MCS-B, respectively.

## 1 Introduction

This paper primarily focuses on addressing the lock-unlock problem under high contention. Despite the significant increase in the number of cores in a *central processing unit* (CPU), a fully shared cache memory system can limit the bandwidth of the cache memory, creating a performance

bottleneck. To overcome this issue, CPUs can maintain private caches, and processors sharing these private caches are referred to as *Cache Coherent Non-uniform Memory Access* (NUMA) processors (abbreviated as ccNUMA).

Spinlocks and atomic operations are provided to ensure the coherency of shared data in the cache, and programs access shared data in *critical sections* (CS) [5, 6]. However, minimizing data access latency is a crucial issue that can significantly impact CPU performance in accessing shared data in ccNUMA [12, 13]. This depends on the topology of the *Network on Chip* (NoC) and the movement of data between caches, which is triggered by tasks executing in the CPU.

When multiple tasks compete to enter a CS, granting the closest task to the one that just released the lock access can reduce data access latency. However, this can still be costly as core-to-core transmission latencies vary in a CPU [14]. Additionally, allowing the core with the shortest transmission latency to enter the CS may lead to adjacent cores having exclusive access, leading to poor throughput [41].

Inter-core communication limits multicore processor scalability [19, 20]. Transmission latency can be fixed or distance-dependent. While monolithic die processors such as Intel Xeon [2] exhibit similar inter-core communication latency, *Multi-Chip-Module* (MCM) processors like AMD EPYC [2] and Apple M1 Pro [2] use MCM technology to increase the number of cores on a processor affordably and at scale. Next-generation Intel Xeons also use MCM [3], but MCM processors may have varying transmission latency between and within chips.

NUMA-aware spinlocks [25–31] enable cores from the same “NUMA node” to enter the CS in batches. This approach is suitable for multi-core processors, such as AMD EPYC, that have different transmission latencies. We can minimize handover costs by dividing the cores in a multi-core processor into mini-nodes, such as the east and west parts shown in Figure 1, and using a NUMA-aware approach to schedule them. However, transferring locks between cores in a mini-node is not considered in these algorithms. A layered approach (e.g., cohort [25]) can address this, but using too many layers

<sup>1</sup>The source codes of RON can be found at <https://github.com/shiwulo/ron-osi2023>.

(e.g., C-TKT-TKT-TKT) can make spinlocks complex and expensive. Modern processors have non-uniform computing power [47], where higher computing power implies a greater ability to acquire locks. Batch-based algorithms often set a “maximal batch size” periodically to prevent starvation and maintain fairness. However, reducing the batch size for fairness can decrease performance. Unfair lock allocation causes unbalanced resource distribution and reduces throughput, as discussed in Section 2.3.

The optimization principle for low-cost communication is similar to that of data routing in computer networks. Although computer networks can use complex algorithms to produce the best route, such methods are often too expensive for small CS in multi-core processors. Therefore, we pre-calculate the shortest circular route including all cores, and the spinlock algorithm generates a path of threads to enter the CS according to the pre-calculated route. The “one-way circular shortest routing” shortens the distance between cores, while the “shortest routing” produces a local optimal solution for handover cost. The “one-way” optimizes handover costs by transferring locks in the direction where more threads are waiting, and the “circular” approach limits the number of times a thread waits to enter the CS. Thus, we schedule tasks on the cores to enter CS in a “one-way circular shortest routing” manner to improve the performance and fairness.

This paper makes three main contributions. Firstly, we propose the simple yet effective concept of “one-way circular shortest routing” to solve the fairness and efficiency issues in spinlocks. Secondly, we identify that long-term fairness alone is insufficient for modern processors, which have cores with varying capabilities to grab locks due to differences in computing frequency. Finally, we provide insights on how single-core spinlocks can work alongside multi-core spinlocks without compromising efficiency and fairness.

In Section 2, we discuss the limitations of NUMA-aware spinlocks in minimizing transmission latency in multi-core processors and the negative impact of unfair spinlocks on throughput. Section 3 presents related work in the field. In Section 4, we propose our fair and efficient spinlock algorithm for ccNUMA. Section 5 addresses performance under oversubscription, while Section 6 compares RON with two well-known algorithms. Section 7 discusses the advantages and disadvantages of RON compared to ShflLock and Linux’s qspinlock, and Section 8 concludes the paper.

## 2 Preliminary and Motivation

### 2.1 Data Coherence in ccNUMA

Figure 1 shows an example of the multi-core architecture, in which the connection network of each core group is similar to the *CPU Complex (CCX)* of Advanced Micro Devices, Inc. (AMD). In a multi-core architecture, data stored in the cache memory can be shared among the cores. *Cache coher-*

*ence non-uniform access (ccNUMA)* uses snoop-based and/or directory-based cache coherence algorithms to maintain consistency of shared data in each cache memory [42]. The snoop method broadcasts messages such as “some shared data has been updated”, whereas the directory-based method allows point-to-point communication between nodes. A node can be a core or a group of adjacent cores.

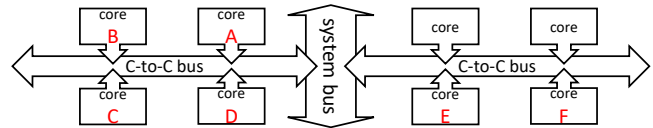


Figure 1: An example NoC architecture of ccNUMA.

### 2.2 Cost of Spinlocks on Multi-core CPUs

We define the *serializing cost* as the cost of allowing multiple threads to have mutually exclusive access to shared data. Serializing costs are divided into “*contention*” and “*handover*”. The contention cost is the cost for determining the next task that can enter the CS. It depends on the data structure and data access method used by a spinlock. For example, the ticket lock [43] is centralized, while MCS spinlock (or called “MCS” for short) [44] is decentralized. In the ticket lock, all threads continuously monitor a variable of the ticket lock, and this can generate a lot of traffics in the NoC. The contention cost also depends on how the threads are granted to enter the CS. The raw spinlock (e.g., GNU’s `pthread_spin_lock` [46], abbreviated as “*Plock*”) relies on the NoC to determine when the first thread can enter the CS. The MCS spinlock [44] allows each thread to wait on a different variable. Therefore, MCS spinlocks prevent atomic operations from triggering excessive bus traffic.

The handover cost depends on the speed of transferring shared data between the lock-holding thread and the successive thread. Because spinlock is a shared data structure, a smaller handover cost can also slightly reduce the contention cost. As the example in Figure 1 shows, the processor is divided by two parts, i.e., the west and east parts. The two parts are connected through a system bus. The handover cost of using the C-to-C bus only is 1. The handover cost between the core and the system bus is related to the distance between the core and the system bus. B, C, and F are far away from the system bus, so the handover cost is 3, and the handover cost of A, D, and E is 2.

In conventional NUMA-aware spinlocks, the order of entering the CS can be arbitrary, for example,  $A \rightarrow D \rightarrow B \rightarrow C \rightarrow F \rightarrow E$ . Since A, B, C, and D have the same communication cost, they belong to the same group (i.e., mini-node). The same goes for E and F. The handover cost of this order is (A, 1, D, 1, B, 1, C, 3, 3, F, 1, E)=10. This paper proposes to use one-way circular shortest routing to minimize

handover costs. By scheduling the order of entering the CS as  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ , the handover cost is reduced to (A, 1, B, 1, C, 1, D, 2, 2, E, 1, F)=8. In this example, the one-way circular shortest routing improves the performance by 20% (i.e.,  $\frac{10-8}{10}$ ).

### 2.3 Throughput or Fairness? Take Both !

Various locking techniques have been proposed [39] to improve system throughput in varying levels of contention. Lock algorithms such as Test-And-Set (TAS) [46] and Test-and-Test-and-Set (TTAS) [46] can be used in low-contention systems. Algorithms like *C-BO-MCS* [25] and *Shuffling (ShflLock)* [26] were designed to reduce the handover cost under high contention with hierarchical design or local spinning.

Moreover, the fairness is an issue that needs to be considered to better utilize the protected resources. According to the level of fairness, we define fairness as follows:

1. *Probabilistic fairness*: The chance of each task entering the critical section is the same in probability.

2. *Bounded waiting*: The number of waiting tasks does not exceed a certain multiple of the number of tasks.

Take Test-Test-And-Set (TTAS) [46] as an example. The probability of each thread obtaining a lock on a single core processor system is related to the proportion of the CPU time that the thread can acquire. In such a situation, the TTAS spinlock satisfies probabilistic fairness. Currently, GNU's Pthread library use TTAS spinlock to implement `pthread_spin_lock()`.

A spinlock algorithm is conformed to bounded waiting when it can limit the number of times that other tasks are inserted before a specific task. Ticket lock and MCS [44] are bounded waiting spinlocks. Both of them are based on first-in-first-out (FIFO) mechanism. Although FIFO allows all tasks to enter CS in a fair manner, FIFO also limits the performance of spinlocks on multi-core/NUMA machines. This is because FIFO cannot shorten the data transmission latency.

Most NUMA-aware spinlocks algorithms balance performance and fairness by preventing threads from waiting too long, but some cores may have higher computing power than others due to differences in manufacturing processes [47]. The slight difference in speed will result in the core with the advantage always being able to acquire the lock successfully. Just like in a 100-meter race, the one who gets first place is always Jamaica's Usain Bolt, even though he is only 0.1 seconds faster than the second-place runner. In modern multi-core processors like the AMD 2990WX, some cores have significantly higher lock acquisition capabilities than others. For instance, the lock acquisition capability of cores 0-7 is 20.6 times greater than that of cores 8-31 (refer to Section 6.2.2). As a result, conventional NUMA-aware algorithms may not be able to ensure equal access to the critical section for all threads/cores within a reasonable period.

With joint consideration of both throughput and fairness, we propose a spinlock method that creates one-way circular shortest path and uses this path to minimize the handover cost and ensure bounded waiting time.

### 3 Related Work

While TTAS spinlock [46] is a simple method to implement POSIX spinlocks in GNU (abbreviated as "Plock") and ensures the consistency of shared data, it is unfair because it tends to provide locks to neighboring cores [21]. Unfairness doubles the execution time of a multi-thread program and causes starvation as shown in [41]. It also increases the variability of latency, making it difficult to guarantee the service quality. The non-scalability of Plock is another serious problem. As shown in [22, 23], although most critical sections are short, increasing the number of cores can cause a system to collapse due to non-scalable locks.

Cohort [25] is a software framework that can combine two NUMA-oblivious locks into a scalable NUMA-aware lock. NUMA nodes compete for the global lock, and unless all threads on the NUMA node leave the CS, the NUMA node will not release the lock. Therefore, threads belonging to the same NUMA node are grouped to enter the critical section, reducing handover costs. Shuffle lock [26] and CNA [27] also use grouping to improve performance. Both are suitable for use with a Linux kernel. However, they cannot effectively reduce the latency of data transmission nor avoid unfairness in a multi-core processor. To obtain good performance under the more complex NUMA architecture, HMCS [28] is based on the concept of Cohort [25] and changes the number of lock levels from 2 to 4. The AHMCS [29] and CLoF [48] algorithms include a mechanism for managing contention and multiple locking methods, allowing different locking methods to be used in different situations. CST-semaphore and CST-mutex locks are applicable to NUMA that support parking [31].

Only dedicated threads or the threads currently holding the lock can execute the code of the CSs of each thread in [32]. In [33], the researchers further proposed turning CSs into an asynchronous execution. Although these methods can optimize data access latency to global data, they take longer to access local data because the code of a CS executes on a specific core.

Programmers can optimize software to better utilize the NoC of ccNUMA when the software uses data-level parallelization and pipe-lining [7, 34, 37, 38]. Stefan Kaestle et al proposed broadcast trees [4] to reduce the communication cost of NUMA machines. However, for multi-threaded programs that use locking mechanisms to protect shared data, these methods may not be suitable.

### 4 Routing On Network-on-Chip (RON)

In Section 4, we introduced RON, a NUMA-aware algorithm that is specifically designed for highly competitive and multi-core environments. In Section 5, we combined RON with simple spin locks, such as plock or ticket lock, to achieve scalability when the number of threads exceeds the number of cores. As most applications typically have more threads than cores, we utilized the RON-plock combination algorithm in our application-level benchmarks.

## 4.1 The Idea

In this section, we propose a design called *Routing On Network-on-Chip* (RON) that aims to minimize the handover cost between cores with low contention cost while ensuring fairness in scheduling the threads waiting to enter a CS. Minimizing handover costs can also improve the efficiency of atomic operations, which are based on atomic operations, which rely on cache coherence protocols (such as snoop+dictionary) on multi-core systems. This, in turn, can improve the performance of locks that suffer from contention.

We propose a concept of reducing the total handover cost by scheduling threads waiting to enter the CS in a specific order. This order can be compared to a train passing through all stations. The ownership of the lock is like the train, and each core is like a station. All waiting threads acquire the lock ownership in order, reducing the total handover cost. Engineers optimize train tracks to pass through all stations in the most efficient way possible, even though the route from station A to station B may not be the shortest. Please note that the train track is a *one-way circular route*. Similarly, we define a global schedule for all cores with waiting threads in the system based on the minimum total handover costs, instead of determining the scheduling order using handover costs alone. One-Way Circular Routing can often achieve global optimization. By minimizing total handover costs, we can also improve the efficiency of atomic operations, thereby improving the performance of locks that suffer from contention.

Since the code of spinlocks cannot be too complicated, it is impractical to dynamically calculate the priority of threads waiting to enter the CS. We assume that there is a thread on each core waiting to enter the CS, and then pre-calculate an optimal lock transfer path. The pre-calculate lock transfer path called the *Traveling Salesman Problem Order* (TSP ORDER) of the cores with an efficient TSP algorithm [40]. For the same processor model, the TSP ORDER is the same. RON follows the TSP ORDER to let threads that want to access shared data enter the CS one by one.

To find the TSP ORDER for a multi-core processor, we created a benchmark program to calculate the transmission latency between cores (see Section 6.2.1). Using this information, we built a fully connected weighted graph of cores and solved the TSP problem with a widely-used algorithm [40]. This allowed us to obtain the TSP ORDER that passes through all cores in the graph, which we use for lock ownership trans-

fer to reduce the handover cost with low contention cost.

## 4.2 The Algorithm

Algorithm 1 presents the RON procedure for one spinlock. We use an array-based method and assume that each core has at most one thread. This method can achieve higher performance under high load compared to using a linked list (similar to MCS [44]). For each spinlock, the array-based RON not only has a “wait flag” for each core, but also places wait flags of adjacent cores, so as to increase the cache efficiency. The data structure of RON is similar to queue spinlock [24] and Linux’s qspinlock with constant space complexity. However, queue spinlock [24] cannot handle the situation in which there are more threads than cores (i.e., oversubscription). In the case of oversubscription, Linux’s qspinlock does not support all tasks to enter CS in the FIFO order to guarantee bounded waiting. Note that we will introduce how to support oversubscription based on an array-based RON in Section 5. It should be noted that RON is a heuristic algorithm and can provide decent solutions but cannot guarantee optimal solutions. The worst case of RON occurs in low contention scenarios where multiple cores access the same memory locations. To mitigate this issue, cache prefetching can be used to predict and fetch the data, reducing the number of cache misses and improving performance.

The first four lines of Algorithm 1 define the variables:

- `NUM_core`: This variable indicates the total number of cores on the system. It is a *system-scope variable*.
- `TSP_ID_ARRAY[]`: This array stores the mapping of each core ID to its corresponding “TSP ORDER ID” (i.e., `TSP_ID`), where `TSP_ID` is the lock transfer order of a core. When a lock is transferred to a core, the thread on this core can be checked to see whether it can enter the CS. This is a *per-process variable*, and each process can have its own routing path (TSP ORDER) because each process owns a different number of cores and can have a different TSP ORDER
- `TSP_ID`: This is the “TSP ORDER ID” of a core, and each thread has its “local version” of `TSP_ID`. Thus, each thread on a different core will get a different value when it accesses the `TSP_ID`. This is a *per-thread variable*. We used “thread\_local”, a C11 keyword of C language, to declare per-thread variable in Algorithm 1 (Line 2).
- `InUse`: If this is “false”, there is no thread in the CS. This is a *per-lock variable*.
- `WaitArray[]`: This array is to indicate which cores’ threads are waiting to enter the CS protected by this lock. When a thread wants to enter a CS, its corresponding `WaitArray[TSP_ID]` is set to 1. When the other threads set their corresponding flag in `WaitArray[]` to



0, the thread can enter the CS. *This is a per-lock array.* In Section 7, we will provide an algorithm for sharing `WaitArray[]` between different locks.

In Algorithm 1, Lines 3 and 5–8 initialize the variables. Line 5 uses `getcpu()` to get the core ID of the running thread, and uses the core ID to get the TSP ID of the core by looking up the `TSP_ID_Array[]`. The `spin_lock()` in Line 10 informs other threads that the caller thread wants to enter the CS. When no other thread is in the CS, the caller thread can enter the CS (Lines 14-16). Otherwise, it waits for the previous thread in the TSP ORDER to leave the CS (Lines 12-13). Because “checking whether there is no thread in CS (lines 19-21)” and “setting `InUse` (line 22)” cannot be executed atomically, it is necessary to simultaneously check `InUse` and `waitArray[TSP_ID]` in a while loop. Additionally, Line 14’s `cmp_xchg()` uses TTAS, a technique commonly used in spinlock implementation to reduce coherence traffic on the cache line.

---

**Algorithm 1** The RON Algorithm

---

```
1  int TSP_ID_ARRAY[NUM_core]; /*per-process*/
2  thread_local TSP_ID; /*thread-local-storage*/
3  atomic_bool InUse=false; /*per-lock*/
4  atomic_int WaitArray[NUM_core]; /*per-lock*/
5  TSP_ID = TSP_ID_ARRAY[getcpu()]
6  void spin_init()
7      for (each element in WaitArray)
8          element = 0;
9  void spin_lock()
10     WaitArray[TSP_ID]=1;
11     while(1)
12         if (WaitArray[TSP_ID]==0)
13             return;
14         if (cmp_xchg(&InUse, false, true)):
15             WaitArray[TSP_ID] = 0
16             return;
17 void spin_unlock()
18     for (int i=1; i<NUM_core; i++)
19         if (WaitArray[(i+TSP_ID)%NUM_core]==1)
20             WaitArray[(i+TSP_ID)%NUM_core]=0;
21         return;
22     InUse=false;
```

---

The `spin_unlock()` in Lines 18-21 finds the next thread that wants to enter the CS. Lines 18-20 treat `WaitArray[]` as a circular queue. From the next position of the caller thread (where `i` is between 1 and `NUM_core`.), it searches for the first thread wanting to enter the CS. Because the thread that wants to enter the CS will set `WaitArray[]` based on its `TSP_ID` (Line 10), the first thread found in the loop of Lines 18-20 is the next thread in the TSP ORDER. In Line 20, `WaitArray[]` of the next thread is set to 0, and the next thread leaves `spin_lock()` (Lines 12-13) to enter the CS. If no thread is waiting, `InUse` is set to false (Line 22).

### 4.3 Correctness

A method must satisfy the following three conditions to ensure the correctness of a CS: (1) mutual exclusion, (2) progress, and (3) bounded waiting. At a minimum, the algorithm used in a software system must satisfy conditions 1 and 2. For instance, GNU’s `pthread_spin_lock` satisfies only conditions 1 and 2, while RON satisfies all three. However, we provide proof of bounded waiting only due to space limitations.

*Bounded Waiting:* We will prove that the maximum number of waits is the number of threads when each core has at most 1 thread. Each core has a unique `TSP_ID`, and these `TSP_ID`s of cores form a circular queue. RON allows all threads to enter a CS in the order of the TSP ORDER. In the worst case when thread `X` is ready to enter a CS, all threads on the cores whose TSP ORDERS are before the core of thread `X` want to enter the CS. Assuming that the total number of threads is “num,” thread `X` needs to wait for (num - 1) threads to leave the CS. In Section 5, RON can support multiple threads on a core. In this case, the maximum number of waits is also the number of threads minus one.

### 4.4 An Example

RON does not prioritize threads for entering the CS based on arrival order, but instead uses the TSP ORDER of each core. While this approach may not generate the optimal solution in all cases, it provides a heuristic algorithm that works efficiently. Let us use the CPU architecture of AMD as an example to illustrate the mechanism of RON. As Figure 2 shows, two CPU Complexes (CCXs) are connected by two point-to-point buses. Each CCX contains four cores that are fully connected by a high-speed network. First, we assume that the TSP ORDER of the cores is  $3 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 4$ . The TSP ORDER of a core can be obtained by `TSP_ID_ARRAY[]`. Taking core 3 as an example, we can find that its TSP ORDER is 0 in `TSP_ID_ARRAY[3]`. We also assume that at time  $t_0$ , the thread on core 3 is ready to enter the CS. Therefore, `InUse` is set to true (Line 14 in Algorithm 1) and this thread on core 3 enters the CS. Then, all entries of `WaitArray[]` in the graph are null (value 0) at time  $t_0$ . At time  $t_1$ , the threads on cores 1, 5, 2, and 6 arrive and are in the Lock Session (LS). Taking the thread on core 1 as an example, its TSP ORDER is `TSP_ID_ARRAY[1] = 2`. Therefore, `WaitArray[2]` is set to 1 (Line 10), and the thread waits for either `WaitArray[2]` (Line 12) or `InUse` (Line 14) to become 0. The `TSP_ID`s of cores 1, 5, 2, and 6 are 2 (`TSP_ID_ARRAY[1]`), 4 (`TSP_ID_ARRAY[5]`), 3 (`TSP_ID_ARRAY[2]`), and 5 (`TSP_ID_ARRAY[6]`), respectively, and their `WaitArray[]` values are set to 1 accordingly.

At time  $t_2$ , the thread on core 3 leaves the CS. Because the TSP ORDER of core 3 is 0, the search will start from next TSP ORDER (i.e., `TSP_ID` 1 in this case). Thus, the

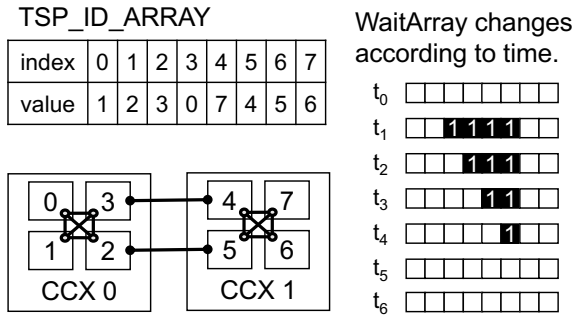


Figure 2: An example of RON.

value of `WaitArray[1]` is examined (Lines 18-21), and the first “1” appears in `WaitArray[2]`. Therefore, the thread in core 3 sets `WaitArray[2]` to 0. Since the thread in core 1 has been waiting for `WaitArray[2]` to become 0 (Line 12), it can now enter the CS. Similarly, at times `t3` and `t4`, the threads in cores 2 and 5 enter the CS, respectively. At time `t6`, the thread in core 6 wants to leave the CS, so it finds all entries of `WaitArray[]` equal to 0. Therefore, it sets `InUse` to false (Line 22).

In this example, we assume that the handover cost within the same CCX is 1 and that across CCXs is 3. If the CS is entered in the FIFO order (3, 1, 5, 2, 6) as in MCS and Ticket, the total handover cost will be  $1 + 3 + 3 + 3 = 10$ . However, according to RON, it will be entered in the order 3, 1, 2, 5, 6, so the total handover cost is only  $1 + 1 + 3 + 1 = 6$ .

## 5 More Threads than Cores

In real applications, there may be a situation where the number of running threads is more than the number of cores. We call this oversubscription. RON approach proposed in Section 4.2 cannot handle oversubscription. In this section we propose two methods to solve this problem: RON-ticket and RON-plock. The former provides better fairness (i.e., bounded waiting), while the latter provides better performance and probabilistic fairness. In the following, we first point out that it is not necessary to run all threads with NUMA-aware spinlock algorithms in Section 5.1. By utilizing this observation without violating fairness, we present our solution on supporting oversubscription in Section 5.2.

### 5.1 Lock Contention Problems on a Core

In oversubscription, multiple threads can run on a single core, which differs from the situation where competing threads are spread across multiple cores. In Figure 3-(a), `Thr1` to `Thr4` correspond to `core1` to `core4`. `core1` and `core2` belong to NUMA `node1`, and the other cores belong to `node2`. If Plock is used and `Thr4` releases the lock, `Thr3` has more probability of entering the CS because `Thr3` and `Thr4` are in the same node.

When `Thr3` and `Thr4` continue to request entering the CS, then `Thr1` and `Thr2` may not have the opportunity to enter the CS. In ticket lock, these threads enter the CS in FIFO order.

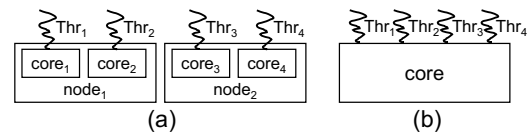


Figure 3: Threads on NUMA nodes vs. threads on a core.

Figure 3-(b) is the same as Figure 3-(a), but all threads belong to the same core. Taking Linux as an example, the execution order of threads on the same core depends on the scheduler. If Plock is used, when `Thr4` (abbreviation for thread 4) releases the lock, the next task that enters the CS is the task executed after `Thr4`. Therefore, the chance of `Thr1` to `Thr3` entering the CS is proportional to their chance of getting CPU time. Because RON guarantees that each core has an equal chance of obtaining the lock, the fairness of threads obtaining the lock on different cores depends on whether the scheduler is fair. The fairness of the ticket lock is the same as in the example shown in Figure 3-(a).

### 5.2 RON with Oversubscription Support

In RON, the element in `WaitArray` indicates whether a thread on that core is waiting to enter the CS. In this section, each element of `WaitArray` indicates how many threads are waiting for the lock on that core (for RON-ticket and RON-plock) and the order in which they enter the CS (for RON-ticket).

The RON-ticket is given in Algorithm 2. Each lock has an array consisting of the elements corresponding to each core and the elements consist of two variables: grant and ticket.

Each core has its own `nWait` variable, which behaves more like thread-local storage. When a thread is waiting to enter the CS from the LS, it uses the `atomic_fetch_add(nWait, 1)` operation to check whether there is a thread in the CS or not. This operation is performed on the `nWait` variable of the core that the thread is running on. If no thread is in the CS, then the waiting thread can enter. To enter the CS, the thread uses the `atomic_fetch_add(ticket, 1)` operation to set the `l_ticket` variable (Line 6). The thread then waits on the while loop (Lines 7-10) until it is its turn to enter the CS. If the thread is not the next thread that should enter the CS of the core (that is,  $grant - l\_ticket \neq 1$ ), the thread releases the CPU (Lines 8-9) and tries again later. When a thread leaves the CS, it first checks to see if there is any waiting thread (Line 13). If there is a waiting thread, it searches for a core with a waiting thread (Lines 14-19). Once a core with a waiting thread is found, it increases the grant of that core by 1 (Line 17), allowing the waiting thread to enter the CS.

The RON-plock is shown in Algorithm 3. Each lock has an array consisting of the elements corresponding to each core and the elements consist of two variables: `numWait` and `lock`.

---

**Algorithm 2** The RON-ticket Algorithm

---

```
1 struct TicketLock {grant=0, ticket=1;}
2 atomic_int nWait=0; //per-lock variable
3 TicketLock WaitArray [NUM_CORE]; //per-lock
  variable
4 TSP_ID = TSP_ID_ARRAY[getcpu()]
5 void spin_lock()
6 if(atomic_fetch_add(&nWait, 1) == 0) return;
7 l_ticket = atomic_fetch_add(&WaitArray[TSP_ID
  ].ticket, 1);
8 while(1)
9 if(WaitArray[TSP_ID].grant-l_ticket≠1)
10 sched_yield();
11 if(l_ticket==WaitArray[TSP_ID].grant) return;
12 void spin_unlock()
13 if(atomic_fetch_sub(&nWait, 1) == 1) return;
14 next = (TSP_ID+1)%NUM_core;
15 while(1)
16 if(WaitArray[next].grant - WaitArray[next].
  ticket ≤ -2)
17 atomic_inc(&WaitArray[next].grant, 1);
18 return;
19 next = (next+1)%NUM_core;
```

---

Each thread that wants to enter the CS must use `atomic_inc()` to set the `numWait` to which it belongs. When the lock of a core is `HAS_LOCK`, the thread currently executing on the core can enter the CS (Lines 7-8). To increase cooperation between the lock-unlock algorithm and the scheduler, `yield()` can be used when multiple threads are executing on a single core. Although `yield()` is a system call and can have overhead equivalent to `futex()`, for user-mode threads, it can be a user-mode function that transfers control to other threads on the same core. When the thread leaves the CS, it searches for the next core whose `numWait` is not equal to 0 and sets the lock of that core to `HAS_LOCK`. If necessary, `yield()` can be used again to allow other waiting threads on the same core to proceed. The proof of correctness is shown in the supplementary material.

## 6 Performance Evaluations

### 6.1 Evaluation Platform and Settings

In the performance evaluation experiments, we used a AMD Threadripper 2990WX with 64 cores (/32 physical cores) with a GNU/Linux operating system. The kernel version was 5.4. The compiler used `gcc-9.3` with the optimization parameter `-march=znver1 -O3`, which enabled `gcc-9.3` to perform the optimization for the Threadripper microarchitecture. All experiments were conducted 100 times, and their results were averaged. The source codes of RON in this section can be found at <https://github.com/shiwulo/ron-osdi2023>.

For a more complete comparison with other methods, we used the LiTL framework [39]. We compiled RON as a shared library. We wrote Algorithm 3 into a program that is compiled

---

**Algorithm 3** The RON-plock Algorithm

---

```
1 struct PLock {numWait=0, lock=MUST_WAIT;}
2 atomic_bool InUse=false; //per-lock variable
3 PLock WaitArray[ NUM_core]; //per-lock variable
4 void lock()
5 atomic_inc(&WaitArray[TSP_ID].numWait);
6 while(1)
7 if (cmpxchg(&WaitArray[TSP_ID].lock, HAS_LOCK,
  MUST_WAIT))
8 return;
9 if (cmpxchg(&InUse, false, true))
10 return;
11 void unlock()
12 atomic_dec(&WaitArray[TSP_ID].numWait);
13 for(int i = 1; i < NUM_core+1; i++)
14 if (WaitArray[(TSP_ID+i)%NUM_core].numWait>0)
15 WaitArray[(TSP_ID+i)%NUM_core]=HAS_LOCK;
16 return;
17 InUse=false;
```

---

with LiTL. By using `LD_PRELOAD`, RON can be compared with other methods on different benchmarks. AMD Threadripper is a chip-NUMA. There are four dies in the chip. Each die has two CCXs, each of which has four cores. Moreover, the Linux `numastat` command shows that 2990WX has 4 NUMA nodes.

The cache coherence protocol operates at the cache line granularity, which means that low latency also implies high bandwidth. Therefore, the transmission latency obtained from the experiments shown in Figure 4 not only informs the design of inter-core locking algorithms but also provides insights into the underlying hardware’s performance characteristics. By profiling the inter-core latency, an operating system can optimize the lock-unlock algorithms accordingly. Furthermore, detailed microarchitecture information about the NoC from CPU vendors can lead to even better performance. In calculating the transmission latency from core X to core Y, we make core X read 100 integers (2 cache lines in this case) from DRAM, and then we calculate the time for core Y to read the 100 integers from core X’s cache. As 2990WX is a ccNUMA architecture, Y will read 100 integers from X’s cache. It should be noted that not all dies on a 2990WX are the same due to differences in the manufacturing process. AMD puts the best cores on die 0, which means that the transmission latency of die 0 is lower. AMD and Intel support “AMD Turbo Core” and “Turbo Boost Max 3.0”, respectively. The operating system can learn how to make better use of the CPU by being aware of the best die. However, traditional NUMA-aware spinlocks cannot achieve the fairness they claim in such processors, which will be discussed in Section 6.2.2.

After obtaining the handover time (i.e., transmission speed) for each pair of cores, we used Google OR-Tools [40] (A solver for NP-complete problems, providing a usable solution.) to determine the TSP ORDER for the cores as shown in Figure 4. We see that the TSP-ORDER first visits all the cores

in the same CCX and then the CCXs on the same die. Finally, the TSP-ORDER visits each die in the clockwise direction. Then, we generate TSP\_ID\_ARRAY[] according to the TSP ORDER for Algorithm 1.

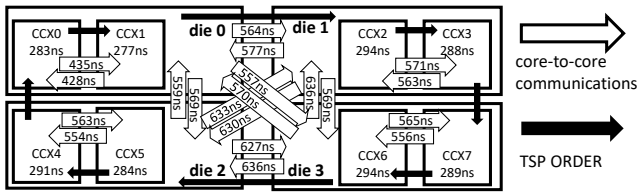


Figure 4: The core-to-core communication latencies and TSP ORDER of AMD 2990WX

We used microbenchmarks in Section 6.2 to analyze the characteristics of the algorithm, and used general benchmarks in Section 6.3 to understand the performance under various usage scenarios. We compared RON with the following algorithms. Please note that what we describe below are the performance characteristics of each algorithm, not the implementation details.

1. Plock: The GNU Pthread’s spinlock. A thread that intends to enter a CS will test the lock until its value equals to 0. When a thread leaves a CS, it sets the lock to 0. The first core that observes that the lock is 0 can enter the CS. The closer to the core the lock is released, the more likely it is for the core to enter the CS.
2. Ticket: This method allows each task waiting to enter the CS to have a “ticket” number. The thread waits until the “grant” is equal to its ticket number. The wait loops of all waiting threads use atomic instructions to continually query the value of the “grant”, which consumes limited NoC bandwidth.
3. MCS: Because all tasks waiting to enter the CS are queued in a linked list, when a thread leaves the CS, it only needs to set the “wait flag” of the next task to false. Setting the wait flag of next thread is more efficient than multicasting when the CPU supports a directory cache coherence algorithm. MCS does not optimize the interconnect latency in multi-core architectures.
4. C-BO-MCS: The thread should first acquire the MCS lock of the NUMA node to which the thread belongs. Then, it must compete with threads on other NUMA nodes to obtain a back-off lock. If a core neighbors to the core that obtains the C-BO-MCS lock, it has a higher priority to enter the CS. With this method, threads belonging to the same node can be grouped together to reduce handover costs.
5. ShflLock (also known as Shuffle Lock): This also uses grouping to improve performance. Shuffle can specify

that a thread in the queue is responsible for shuffling. However, when the task that is allowed to enter the CS is shuffling the queue, the thread cannot enter the CS immediately and system performance may decrease.

## 6.2 Microbenchmarks for Quantitative Analysis

### 6.2.1 Evaluation Platform and Settings

Here, we analyzed each spinlock method in a quantitative manner through a controllable microbenchmark. In each set of experiments in this section, each thread is bound (i.e., `sched_setaffinity()`) to a hardware thread and executes Algorithm 4. Because we have SMT (Simultaneous multi-threading) enabled, there are 2 hardware threads per core. The total number of software threads is 64. In the while loop (Lines 2–9), a thread in the *lock section* (LS) (Line 3) requests entry into the *critical section* (CS) (Lines 4–5). After the thread enters the CS, each entry in `SharedData` is read and written, and the lock is released into the *unlock section* (US) (Line 6) when the thread leaves the CS. The `clock_gettime()`, defined in the POSIX.1-2001 standard, is called in the *non-critical section* (nCS) (Lines 7–9) until the elapsed time of the nCS exceeds the value of `nCS_size±15%` in Line 9. We first evaluate the throughput (Figure 5) and fairness (Figure 6) of each algorithm, and then analyze their efficiency in terms of handover (Figure 7) and contention (Figure 8). Please note that in these 4 experiments, except for adding the code for measuring time (i.e., `clock_gettime()`) and the code for statistics, the experimental parameters are the same.

---

#### Algorithm 4 Testing Program and Measurements

---

```

1 void thread():
2     while(1):
3         spin_lock(); //LS
4         for (each element in SharedData): //CS
5             element = element + 1; //CS
6         spin_unlock(); //US
7         t = clock_gettime(); //nCS
8         //syscall overhead, rdtscp implement in userspace
9         while(clock_gettime()-t > nCS_size*rand(0.85~1.15));

```

---

### 6.2.2 Results of Microbenchmarks

As shown in Algorithm 4, a shorter nCS implies a heavier workload because the lock request rate is higher. The upper and lower parts of Figure 5 are the performance when the contention is low and high, respectively. RON can provide the best locking efficiency in both cases. Under low load conditions (`nCS = 400K~120K`), the performance of Plock



is second only to RON. When nCS is lower than 40K, the performance of Plock drops rapidly when the load is heavy. The performance curves of MCS and ShflLock are similar, which may be because they queue tasks waiting for CS in a linked list.

ShflLock and C-BO-MCS perform worse than MCS in some cases (i.e., nCS > 10K). This is because these two algorithms implicitly treat the handover costs between cores belonging to the same die as equal. Therefore, they cannot optimize the handover costs of different cores in different dies (Please see the example in Section 2.2). Further, since the difference in communication cost between 2990WX cores is only 3.13 times at most, an algorithm with too-high time complexity reduces the benefits that can be obtained. Because RON uses a pre-calculated TSP ORDER that is optimized for all cores, it can achieve higher performance at a lower cost. According to our simulation results (in the supplementary material), ShflLock and C-BO-MCS performs better when the transmission latency between cores on the same chip/die is almost the same. In other situations, RON performed best.

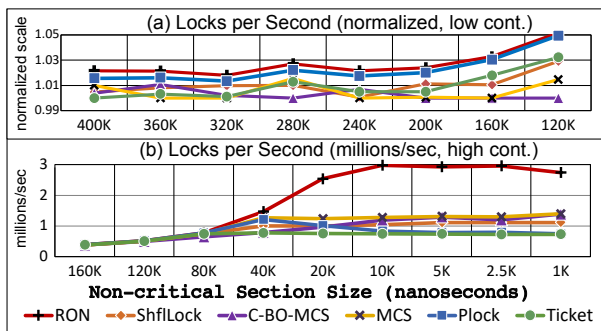


Figure 5: Locks per second under different loads

Figure 6 shows the number of locks acquired by each core in the case of short-term (1 second) and long-term (10 seconds). The lower the coefficient of variation (CV), the better the fairness. In Figure 6, we see that RON, MCS, and Ticket perform equally well, in terms of CV. That is almost equal to 1%. In long-term fairness, when non-critical section (nCS) < 80K ns, the CV of Plock starts to rise. When nCS < 20K ns, the CV of ShflLock and C-BO-MCS both starts to rise. In order to better understand the performance of spinlock algorithms in long-term fairness, we let the ShflLock, C-BO-MCS, and Plock execute for 1,000 seconds with nCS = 10K and their CVs are 35%, 71% and 96%, respectively. *Fairness factor* is described by Dice et al. [25]. It is the most common metric to measure fairness. The value of fairness factor is between 0.5 and 1. A complete fair spinlock's factor is 0.5 and a complete unfair spinlock's factor is 1. The fairness factor of the ShflLock, C-BO-MCS and Plock are 0.68, 0.85 and 0.8, respectively.

In terms of software design, each thread in Plock competes fairly for locks. C-BO-MCS is based on two fair spinlocks, namely backoff [44] and MCS. ShflLock allows threads on

the same node (i.e., die) of the lock holder to elevate their positions in the queue for a limited number of times. Note that it is difficult to analyze in detail why these algorithms do not meet long-term fairness perfectly, so only Plocks is analyzed to provide insights into the interaction between multicore processors and spinlocks.

In the past, the multi-core processor had to execute at a frequency that all cores could run correctly. The worst core determines the maximum clock frequency that a multi-core processor can run. Now each core can run on its highest frequency [47]. According to the experimental results of the Plock with nCS=10k, the ability of cores 0-7 and 32-39 to obtain locks is 20.6 times that of cores 8-31 and 40-63. Therefore, we roughly conclude that when the load becomes heavier, algorithms that meet long-term fairness may not achieve the expected fairness on modern multi-core processors. [47].

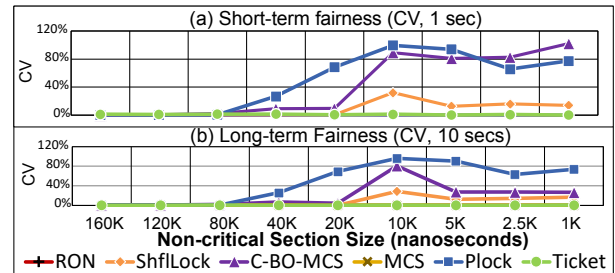


Figure 6: Long-/short- fairness of different algorithms.

The experimental parameters of Figure 7 are the same as Figures 5 and 6, but we changed the X coordinate from nCS (Line 8 in Algo. 4) to the number of threads waiting to enter CS (i.e., the number of threads in LS, Line 3 in Algo. 4). The more the number of threads waiting, the better the performance of an algorithm that can optimize the handover cost. In Figure 7, the Y axis is the time required to access the shared data. For example, in the case of RON under load nCS = 10K, the number of threads in LS is 45, and the handover time is 100 ns. Under the same load, the number of threads of C-BO-MCS in LS is 52, and handover time is 190 ns. RON is almost the best spinlock in terms of handover time. With more tasks in the LS, the path formed by each task selected by RON is closer to TSP ORDER, because each core has a higher probability to have a thread waiting for entering the CS. When the number of tasks in LS increases from 0 to 15, the efficiency of accessing shared data doubles (from 210ns to 100ns). When the LS changes from 60 to 62, the efficiency is reduced by 7%. This is the reason for the reduced efficiency when the nCS is 1K in Figure 5.

Plock is slightly better (0.07%) than RON when the lock contention is very low (nCS=120K). Although Plock's handover cost is low, its performance is not good. Since Plock uses `cmp_xchg` (`compare_exchange`) to solve the lock contention problem, The hardware may need to execute `cmp_xchg` continuously until the return value of only one task is equal to true. This wastes the limited bandwidth of

NoC and is time consuming. The handover time of C-BO-MCS and ShflLock is better than MCS. However, these two methods are too complicated, resulting in the performance lower than expected. Both Ticket and MCS arrange tasks to enter the CS in the order of their arrival. Since Ticket does not give each thread a wait flag, all threads will constantly monitor the wait flag, thus consuming a lot of NoC bandwidth. Ticket has the worst handover cost.

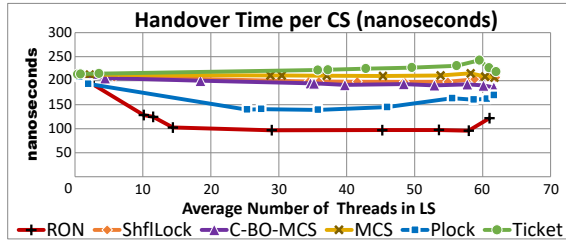


Figure 7: Handover cost and the number of thread in LS.

In Figure 8, we set the size of the shared data accessed in the critical section to 0 (Line 4 in Algorithm 4). The X axis is the number of tasks waiting to enter the CS, and the Y axis is the time of each thread executing one round (including LS, US, CS and nCS, i.e., Lines 2-8). The size of the non-critical section (Lines 7 and 8) ranges from 160K to 1K. Therefore, the main factor in performance is the locking and unlocking efficiency of an algorithm. RON is almost the best algorithm. Its performance is slightly worse than that of Plock (0.2%) when the loading is extremely light. RON has a better performance for two reasons. First, TSP ORDER is pre-calculated. Second, the lower handover cost makes atomic operations more efficient. We use experiments to analyze the efficiency of atomic operations of RON. When we schedule threads to perform atomic operations through TSP ORDER, the efficiency of atomic operations is 1.6 times that of random order.

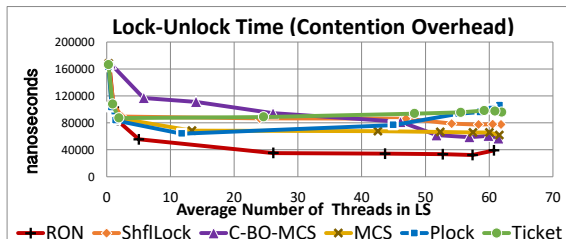


Figure 8: Contention cost.

### 6.2.3 Oversubscribe

In each set of experiments of this section, each thread binds (i.e., `sched_setaffinity()`) to a core and executes Algorithm 4. Each core has at most  $\lceil \text{num\_thread} \div \text{num\_core} \rceil$  threads. Because RON-ticket shares a key property with RON, that is, bounded waiting, RON-ticket was used for performance evaluation in the previous section. In this section,

microbenchmarks are used to evaluate the performance of RON-ticket and RON-plock. In the case of oversubscription, two factors affect performance. The first one is whether the thread holding the lock is scheduled out. Second, if the algorithm specifies the next thread entering the CS, and whether it is scheduled out.

In Figure 9, RON-plock and RON-ticket perform better in the case of overbooking, where the y-axis denotes “millions locks” per second. Although C-BO-MCS(-B) and ShflLock(-B) also support oversubscription, the number of lock-unlock operations per second is dropped quickly.

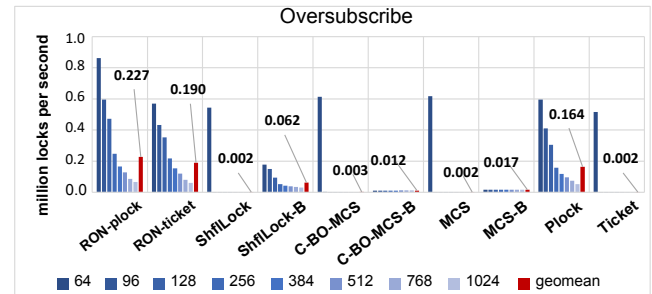


Figure 9: Performance of algorithms under oversubscribe.

RON-plock and Plock use intuitive methods (e.g., test-test-and-set) to solve the problem of oversubscribe. As long as the lock-holder is not scheduled out, Plock will allow a thread to enter CS (it is because that all threads wait on the same variable.). RON-plock is similar to Plock, except that all threads on the next core are scheduled out. Because RON-plock is based on RON, the performance of RON-plock is better than Plock. RON-ticket, ShflLock-B, and C-BO-MCS-B use system calls (i.e., `futex()` and `yield()`) to prevent the thread from spinning meaninglessly. ShflLock-B’s `unlock()` directly wakes up the next thread. However, C-BO-MCS-B’s `unlock()` may wake up all threads that can enter CS. RON-ticket makes the next thread that can enter the CS busy waiting, and other threads on that core enter the sleep state. For the same reason as RON-plock, RON-ticket has better performance.

### 6.2.4 Scalability

In this section, we investigate how algorithm performance changes with an increase in the number of threads used. Our experiment was conducted on the 2990WX, which has SMT technology. During the experiment, each thread accesses 100 integers in the critical section, while the non-critical section takes  $10,000 \pm 15\%$  nanoseconds.

As shown in Figure 10, it indicate that the performance of the RON-plock improves with an increase in the number of threads when the thread count is less than 64. This experiment yields results similar to those in Figure 5 because “executing more threads simultaneously” and “having shorter non-critical sections” both lead to greater competition for entering the

critical section. However, when the number of threads exceeds 64, the performance of these algorithms is determined by their ability to handle the oversubscribe problem.

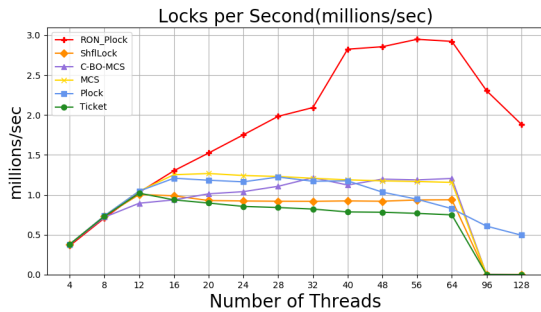


Figure 10: Locks per second under different number of threads

### 6.3 Application-level Benchmarks

We pick five different application-level benchmarks representing different performance bottlenecks. For the consistency of the experiment, the RON implementation here uses RON-ticket, which has identical features to RON (bounded waiting).

#### 6.3.1 LevelDB (Key-value Database)

Here, we used Google’s LevelDB to test the performance of the spinlock. The Horizontal axis of Figure 11 is the algorithm tested, and the vertical axis is the time cost for every operation reported by LevelDB. Because of the difference data scales, “fillsync” is normalized to MCS and others normalized to Ticket.

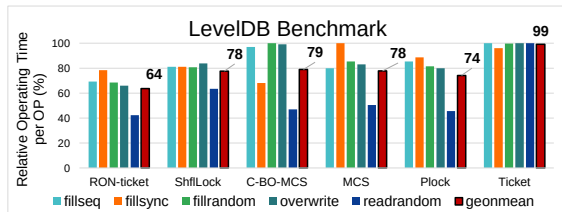


Figure 11: Google’s LevelDB.

We use the db\_bench in LevelDB to evaluate performance with 1 million entries and 64 threads. For each spinlock, fillseq, fillsync, fillrandom, overwrite, and readrandom have been tested. The last one is the geometric average of LevelDB’s 5 tests. RON-ticket, ShflLock and C-BO-MCS are spin locks optimized for ccNUMA or NUMA. Please note that RON-ticket is RON with oversubscribe support and it also satisfies bounded waiting. Compared with ShflLock and C-BO-MCS, the performance of RON is better by 22.1% and 24.2%, respectively.

MCS is slightly better than ShflLock and C-BO-MCS for LevelDB, although the latter two are optimized for NUMA.

This may be because these two algorithms are designed to overcome the huge transmission overhead between two CPUs. However, there is not much difference in the communication cost between the cores on AMD 2990WX. When the load is high, ShflLock and C-BO-MCS may only perform local optimization.

Consider the situation with four NUMA nodes, where ShflLock and C-BO-MCS serve node X, and the load on node X always has a thread waiting to enter the CS. At this time, although there are many threads waiting for the CS on other NUMA nodes to enter, ShflLock and C-BO-MCS tend to let tasks on node X enter the CS. Since RON uses TSP ORDER to arrange the cores to enter the CS, RON does not suffer from the problem of local optimization.

#### 6.3.2 Benchmarks in Different Contention Levels

We applied an additional four different application benchmarks to evaluate the performance of different algorithms. These algorithms are selected from LiTL [39] and cover both high and low contention scenarios. Volrend and Raytrace are from the SPLASH2x benchmark set representing extreme and high levels of contention, respectively. For the extreme level, more than 40 threads are waiting to acquire the same lock instance. For the high level, there are about 10 to 40 threads waiting to acquire a lock. Dedup and Ferret are from the PARSEC3.0 benchmark set and respectively represent pressure on the memory and relatively low levels of contention [39]. In Figure 12, the vertical axis is the elapsed time of the benchmark task (including geomean of LevelDB Figure 11). Because of the different data scales, the numbers are the percentage to where the algorithm performs the worst for each task.

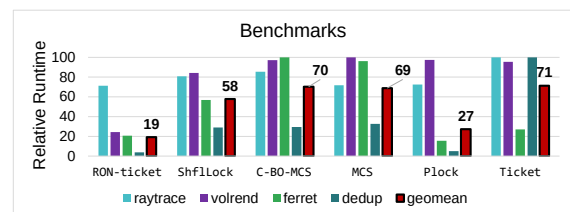


Figure 12: Applications with different contention level.

The bottleneck of Raytrace is a lock contention, protecting a single counter with about a million acquisitions every second. RON-ticket, MCS, and Plock accomplished the task with around 70% of elapsed time. MCS is optimized for multi-core systems with dedicated caches for each core to reduce the overhead of lock contention and well fitted in high level of contention. The code of Plock is not optimized for multi-core. However, the core adjacent to the core that released the lock is more likely to successfully perform the atomic operation compare\_exchange() to acquire the lock. Thus, Plock is implicitly optimized for multi-core platforms.

In the case of extreme levels of contention, the performance of Plock and MCS starts to drop while the ShflLock and RON-ticket can handle the stress. Under extreme level of contention (Volrend), RON-ticket achieved its best performance, taking only 24.3% of the elapsed time of the ticket to accomplish task. The bottleneck of Volrend is the lock contention for protecting different task queues with around 40 threads waiting. This benchmark verified that the RON algorithm generally performs best under higher contention. With more cores possessed by threads spinning for the lock instance, the routing path can massively reduce the handover cost.

However, under low levels of contention, RON-ticket only performs second best in all six algorithms. Ferret is a parallelized software with about 2000 times of acquisition for every second. While RON-ticket uses around 20% of the elapsed time, Plock takes only around 15.6% of the elapsed time of C-BO-MCS, outperforming RON in this specific benchmark. The lower contention of the lock leads to the sparseness of the `WaitArray`, which results in leaping on the routing path and lowers the benefit. Ticket guarantees fairness as threads keeping querying the global variable to know whether they can enter the CS. Ticket fits the task with low level of contention. However, under higher pressure, the bandwidth consumed by lock contention limits the bandwidth that can be used by handover.

Moreover, according to the results of Dedup, RON-ticket and Plock gave low memory pressure. Dedup allocates numerous locks (266k) [39], which puts pressure on the memory if the components of the lock are not reusable. The reusability of components like `WaitArray` and `Get_TSP_ID` gives RON-ticket the ability to handle numerous lock allocations.

In summary, RON algorithms can handle different levels of contention, especially higher levels. With higher contention RON algorithms achieve better performance relatively, but Plock remains a better algorithm for low levels of contention. With reusable components, Both Plock and RON put low pressure on the memory while allocating numerous lock instance.

## 7 RON in GNU/Linux Kernel

### 7.1 Implementation

As shown in the performance evaluation section, RON is more suitable for multi-core computers than methods that support NUMA in user space applications. In this section, we shows whether RON is suitable for Linux kernel. In our implementation, the line of code (LoC) is 47.

In the Linux kernel, the lock-acquire and lock-release are implemented by `queued_spin_lock(struct qspinlock *lock)` and `queued_spin_unlock(struct qspinlock *lock)`, respectively. Both functions have only one parameter, `lock`. By rewriting these two functions, we implement RON in the Linux kernel. We use `*lock` as `InUse` in the RON algorithm (Line 3 in Algorithm 1).

In order to achieve the same space efficiency as `qspinlock`, only one `WaitArray` (Line 4 in Algorithm 1) is in kernel. In user space, a task sets `WaitArray[TSP_ID]` (Line 10 in Algorithm 1) to wait for entering the CS. In the Linux kernel, the task writes the address of `lock` (that is the parameter of `queued_spin_lock`) to `WaitArray[TSP_ID]` for entering the CS. When the thread leaves CS, the thread will check one by one whether there is an element with a value equal to `lock` in `WaitArray`. Therefore, a busy-waiting task is only awakened by the task holding the same `lock`.

If the space of the `WaitArray` has been used up, the other tasks wait on `InUse` (that is `*lock` in kernel space). Tasks waiting for `InUse` do not enter CS in TSP ORDER. This design method is the same as Linux's `qspinlock`, though it is not perfect but good enough (compromise to  $O(1)$  space complexity). In terms of memory usage, RON requires 4 bytes for each lock (that is the size of `struct qspinlock`) and 28 bytes for each core ( $28 \times 64 = 1792$  bytes for AMD 2990WX).

### 7.2 Evaluations

In this section, the Linux kernel version is 5.12.1. We apply the patch of ShflLock into the `qspinlock.c` of Linux. Therefore, in this section, we will compare the performance of the Linux kernel using `qspinlock`, ShflLock and RON. We use a microbenchmark and `db_bench` of Google LevelDB to measure the performance of RON in Linux kernel. In the experiment, we do not use `LD_PRELOAD` to change the behavior of LevelDB. The purpose of microbenchmark is to measure performance under high load conditions. The microbenchmark is implemented by forking 64 child processes, and every child process creates 2048 threads to execute 64 `mmap()` and `munmap()` function calls. We use `strace` to evaluate the time taken for each system call.

As shown in Figure 13, ShflLock doesn't perform well on both microbenchmark and LevelDB. ShflLock is suitable for multi-socket NUMA machines but ours is a single-socket machine. RON performs better than `qspinlock` under high load conditions. In terms of geometric average of microbenchmark, the performance of Linux kernel with RON is 1.35 times that of Linux kernel with `qspinlock`. In terms of LevelDB, RON and `qspinlock` are about the same in terms of geometric average. In these five experiments, the performance of these three algorithms on `readseq` are almost the same. RON performs better than `qspinlock` in `fillsync` because the contention is high. RON and `qspinlock` both have their own strengths.

Intuitively we can combine `qspinlock` and RON to achieve better performance. `qspinlocks` can encode the first two tasks that want to enter CS into the `lock` variable (the parameter of `queued_spin_lock()`). In this way, the performance of `qspinlock` is very good under low contention. The purpose of this experiment is to explore the effectiveness of RON, so we did not use Linux optimization techniques to improve



performance under low contention conditions.

microbenchmark (microseconds)					
system call	mmap	clone	mprotect	munmap	geomean
qspinlock	923	183	252	43	207
ShflLock	2029	206	456	39	294
RON	592	185	264	19	153

(a)

LevelDB						
	fillseq	fillsync	fillrandom	overwrite	readseq	geomean
	MB/sec	op/sec	MB/sec	MB/sec	MB/sec	
qspinlock	20.2	511.4	17.8	17.2	487.3	68.8
ShflLock	7.9	390.5	7.5	7.1	485.3	38.1
RON	18.3	687.1	16.3	15.7	482.7	68.9

(b)

Figure 13: Performance comparisons on Linux kernel.

## 8 Conclusion

We propose a RON spinlock algorithm that delivers locks and data in a one-way circular manner among cores with the awareness of the performance differences of cores, so as to minimize the system-level handover cost and achieve bounded waiting for threads among cores. In particular, “one-way” is for minimized system-level handover cost and “circular” is for bounded waiting of threads to enter CS. In addition, the proposed RON algorithm can also resolve the oversubscription issue without losing its scalability. A series of experiments were conducted to evaluate the efficacy of the proposed algorithm. Compared with ShflLock and C-BO-MCS, the performance of RON in google leveldb has increased by 22.1% and 24.2% respectively. In terms of kernel space performance, compared with using ShflLock, RON can improve the performance of Google LevelDB by 1.8 times.

## 9 Future work

This paper addresses the issue of unfairness caused by different execution frequencies on multi-core processors, as well as the efficiency of inter-core data transfer. The proposed method is particularly suitable for highly competitive scenarios. Although high competition can be a bottleneck for performance, low competition is a more common scenario where simple algorithms often have good performance. Therefore, in future research, we will investigate how to dynamically switch algorithms (such as plock and RON) at runtime. We will also evaluate the performance of RON by implementing it using linked-list methods to offload the runtime of unlocking to the locking process.

## Acknowledgement

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their valuable feedback and suggestions. This

work was supported in part by Ministry of Science and Technology (MOST) of Taiwan under grant nos. 111-2221-E-194-017-MY3, 111-2223-E-001-001, 111-2923-E-002-014-MY3, 111-2221-E-001-013-MY3, and 112-2927-I-001-508.

## References

- [1] L. T. Su, S. Naffziger and M. Papermaster, "Multi-chip technologies to unleash computing performance gains over the next decade," 2017 IEEE International Electron Devices Meeting (IEDM), 2017, pp. 1.1.1-1.1.8, doi: 10.1109/IEDM.2017.8268306.
- [2] Nicolas Viennot (Sep 19, 2022). Measuring CPU core-to-core latency. <https://github.com/nviennot/core-to-core-latency>
- [3] Sally Ward-Foxton (08.19.2021). Intel Brings Chiplets to Data Center CPUs. EETimes. <https://www.eetimes.com/intel-brings-chiplets-to-data-center-cpus/>
- [4] Stefan Kaestle, Reto Acherhmann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, Timothy Roscoe: Machine-Aware Atomic Broadcast Trees for Multicores. OSDI 2016: 33-48
- [5] <https://www.freebsd.org/cgi/man.cgi?query=atomic&sektion=9&format=html>
- [6] Daniel Sorin; Mark Hill; David Wood, “A Primer on Memory Consistency and Cache Coherence,” Morgan & Claypool, 2011.
- [7] Pradip Kumar Sahu and Santanu Chattopadhyay. 2013. A survey on application mapping strategies for Network-on-Chip design. J. Syst. Archit. 59, 1 (January, 2013), 60–76. DOI:<https://doi.org/10.1016/j.sysarc.2012.10.004>
- [8] Rajesh Chopra, Yang-Trung, LinSailesh Kumar, “Generating physically aware network-on-chip design from a physical system-on-chip specification,” US Patents US10218580B2, Application granted in 2019.
- [9] C. Wu et al., “A Multi-Objective Model Oriented Mapping Approach for NoC-based Computing Systems,” in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 3, pp. 662-676, 1 March 2017.
- [10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, “The directory-based cache coherence protocol for the DASH multiprocessor,” In Proceedings of the 17th Annual International Symposium on Computer Architecture (ISoA), Seattle, WA, USA, pp. 148-159, 1990.
- [11] Chinya Ravishankar, James Goodman, “Cache Implementation for Multiple Microprocessors,” in Proceedings of IEEE Computer Conference, pp. 346–350, Feb 1983.



- [12] Ruibo Wang, Kai Lu, and Xicheng Lu. 2009. Investigating transactional memory performance on cc-NUMA machines. In Proceedings of the 18th ACM international symposium on High performance distributed computing (HPDC '09). Association for Computing Machinery, New York, NY, USA, 67–68. DOI:<https://doi.org/10.1145/1551609.1551625>
- [13] R. R. Iyer and L. N. Bhuyan, "Design and evaluation of a switch cache architecture for CC-NUMA multiprocessors," in *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 779–797, Aug. 2000, doi: 10.1109/12.868025.
- [14] K. A. Bowman, A. R. Alameldeen, S. T. Srinivasan and C. B. Wilkerson, "Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors," Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07), 2007, pp. 50–55, doi: 10.1145/1283780.1283792.
- [15] "Ampere® Altra® offers up to 80 cores at up to 3.0 GHz", 80 cores, <https://amperecomputing.com/altra/>
- [16] "AMD EPYC™ 7003 Series Processors scale from 8 to 64 cores", 64 cores, <https://www.amd.com/en/processors/epyc-7003-series>
- [17] "Intel® Xeon® Platinum 8380 Processor (60M Cache, 2.30 GHz)", 40 cores, <https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/platinum.html>
- [18] "Arm-based AWS Graviton2 processors", 64 vCPU, <https://aws.amazon.com/tw/ec2/instance-types/x2/>
- [19] Abdul Naeem, Xiaowen Chen, Zhonghai Lu, Axel Jantsch. "Scalability of relaxed consistency models in NoC based multicore architectures". *ACM SIGARCH Computer Architecture News*. April 2010.
- [20] B. K. Daya et al., "SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 25–36, doi: 10.1109/ISCA.2014.6853232.
- [21] scientiaesthete. 2012 "pthreads: thread starvation caused by quick re-locking", Retrieved June 20, 2019 from <https://stackoverflow.com/questions/12685112/pthreads-thread-starvation-caused-by-quick-re-locking>
- [22] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris and Nickolai Zeldovich, "Non-scalable locks are dangerous", in Proceedings of the Linux Symposium (OLS2012), Ottawa, Canada, July 2012.
- [23] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, Vancouver, Canada, 1–16.
- [24] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for scalable synchronization on shared-memory multi-processors," *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
- [25] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.* 1, 2, Article 13 (January 2015), 42 pages. DOI:<https://doi.org/10.1145/2686884>
- [26] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and practical locking with shuffling. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 586–599. DOI:<https://doi.org/10.1145/3341301.3359629>
- [27] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 12, 1–15. DOI:<https://doi.org/10.1145/3302424.3303984>
- [28] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. *SIGPLAN Not.* 50, 8 (August 2015), 215–226. DOI:<https://doi.org/10.1145/2858788.2688503>
- [29] Milind Chabbi and John Mellor-Crummey. 2016. Contention-conscious, locality-preserving locks. *SIGPLAN Not.* 51, 8, Article 22 (August 2016), 14 pages. DOI:<https://doi.org/10.1145/3016078.2851166>
- [30] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures (SPAA '11). Association for Computing Machinery, New York, NY, USA, 65–74. DOI:<https://doi.org/10.1145/1989493.1989502>
- [31] S Kashyap, C Min, T Kim, Scalable numa-aware blocking synchronization primitives, USENIX Annual Technical Conference, 2017
- [32] LOZI, J., DAVID, F., THOMAS, G., LAWALL, J., and MULLER, G. "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multi-

- threaded Applications.” USENIX Annual Technical Conference ’12.
- [33] David Klaftengger, Konstantinos Sagonas and Kjell Winblad, “Queue Delegation Locking”, IEEE Transaction Parallel and Distributed Systems, vol. 29, no. 3, pp.687-704, March 2018.
- [34] B. K. Joardar, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu and R. Marculescu, "Learning-Based Application-Agnostic 3D NoC Design for Heterogeneous Manycore Systems," in IEEE Transactions on Computers, vol. 68, no. 6, pp. 852-866, 1 June 2019, doi: 10.1109/TC.2018.2889053.
- [35] W. Amin et al., "Performance Evaluation of Application Mapping Approaches for Network-on-Chip Designs," in IEEE Access, vol. 8, pp. 63607-63631, 2020, doi: 10.1109/ACCESS.2020.2982675.
- [36] S. Das, J. R. Doppa, P. P. Pande and K. Chakrabarty, "Monolithic 3D-Enabled High Performance and Energy Efficient Network-on-Chip," 2017 IEEE International Conference on Computer Design (ICCD), 2017, pp. 233-240, doi: 10.1109/ICCD.2017.43.
- [37] C. Wu et al., “A Multi-Objective Model Oriented Mapping Approach for NoC-based Computing Systems,” in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 3, pp. 662-676, 1 March 2017.
- [38] Aryan Deshwal, Nitthilan Kanappan Jayakodi, Biresh Kumar Joardar, Janardhan Rao Doppa, and Partha Pratim Pande. 2019. MOOS: A Multi-Objective Design Space Exploration and Optimization Framework for NoC Enabled Manycore Systems. ACM Trans. Embed. Comput. Syst. 18, 5s, Article 77 (October 2019), 23 pages. DOI:https://doi.org/10.1145/3358206
- [39] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. “Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems,” ACM Transactions on Computer Systems, Volume 36 Issue 1, March 2019.
- [40] “OR-Tools | Google Developers”. Retrieved June, 28, 2019 from <https://developers.google.com/optimization/>
- [41] Jonathan Corbet, “Ticket spinlocks,” Retrieved from <https://lwn.net/Articles/267968/>
- [42] J. Hennessey and D. Patterson, Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2017.
- [43] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with event counts and sequences. Communications of the ACM 22, 2 (1979), 115–123. DOI:https://doi.org/10.1145/359060.359076
- [44] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9, 1 (1991), 21–65. DOI:https://doi.org/10.1145/103727.103729
- [45] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," Proceedings 22nd Annual International Symposium on Computer Architecture, 1995, pp. 392-403.
- [46] Thomas E. Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1, 1 (1990), 6–16. DOI:https://doi.org/10.1109/71.80120
- [47] Btarunr. Windows 10 2H19 Update to Have "Favored Core" Awareness, Increase Single-threaded Performance. online <https://www.techpowerup.com/259688/windows-10-2h19-update-to-have-favored-core-awareness-increase-single-threaded-performance>
- [48] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, Haibo Chen. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, October 2021, Pages 851–865. DOI: <https://doi.org/10.1145/3477132.3483557>





# Userspace Bypass: Accelerating Syscall-intensive Applications

Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou  
Fudan University  
{zhouzhe, 19210240167, 19210240003, zyf}@fudan.edu.cn

Zhou Li  
University of California, Irvine  
zhou.li@uci.edu

## Abstract

Context switching between kernel mode and user mode often causes prominent overhead, which slows down applications with frequent system calls (or syscalls), e.g., those with high I/O demand. The overhead is further amplified by security mechanisms like Linux kernel page-table isolation (KPTI). To accelerate such applications, many efforts have been put in removing syscalls from the I/O paths, mainly by combining drivers and applications in the same space or batching syscalls. Nonetheless, such solutions require developers to refactor their applications or even update hardware, which impedes their broad adoption.

In this paper, we propose another approach, userspace bypass (UB), to accelerate syscall-intensive applications, by transparently moving userspace instructions into kernel. Userspace bypass requires no modification to userspace binaries or code and achieves full binary compatibility. Specifically, to avoid overhead caused by frequent syscalls, kernel identifies the short userspace execution path between consecutive system calls, and converts the instructions in the path into code blocks with Software-Based Fault Isolation (SFI) guarantee. According to our evaluation, I/O micro-benchmark can be accelerated by 30.3 – 88.3%, Redis GET Requests Per Second (RPS) can be improved by 4.4 – 10.8% for 1B – 4KiB data sizes, when the application is executed in a virtualized setting with KPTI turned on. The performance boost will be reduced when KPTI is turned off.

## 1 Introduction

System call (syscall) is widely used by a userspace application to access the resources provided by the hosting operating system (OS) and extensively used for I/O operations. However, syscall could incur prominent performance overhead [43] when mechanisms like Linux kernel page-table isolation (KPTI) [47] are turned on. Arguably, syscall is one of the major performance bottlenecks for applications pursuing high I/O requests Per Second (IOPS), e.g., those requesting over a million IOPS [7].

**Syscall-refactoring approaches.** In the recent literature, there are mainly two streams of work in achieving higher IOPS by changing how syscalls are processed from the I/O path, which we call syscall-refactoring approaches: 1) The first stream of approaches integrate drivers and data processing logic in the same address space by moving data processing logic into kernel [26, 36, 53] or moving drivers responsible for I/O into userspace (kernel bypass) [21, 51]. In this way, the processing logic can directly talk to I/O devices and avoid the overhead caused by the switching between user mode and kernel mode [51]. 2) The second stream of approaches batch syscalls and allow userspace processes to queue multiple I/O requests and issue them together with only one single syscall [43]. However, these solutions require developers to change their code, which is usually a non-trivial task.

**Our approach.** In this paper, we propose *userspace bypass* (or UB for short), which reduces the overhead introduced by syscall-related I/O and achieves *binary compatibility* (i.e., no application code needs to be changed or rebuilt) at the same time. UB is motivated by the observation that applications with high IOPS do not execute many instructions between two consecutive syscalls (see Section 3.1). As a result, we can transparently instrument the instructions between syscalls under pre-defined security requirements (i.e., translating the instructions into sanitized code blocks), and let kernel execute the blocks without returning to userspace. In this way, the overhead caused by consecutive syscalls can be avoided. Figure 1 illustrates this idea.

Yet, a few challenges should be addressed. First, only those instructions that will potentially be executed between *frequently* invoked consecutive syscalls deserve userspace bypass. However, without explicit information provided by the developer, it is difficult to find such syscall sequence. As elevating the instructions to kernel also introduces overhead, the syscalls to be optimized need to be carefully chosen to offset such overhead. Second, malicious applications may exploit UB to steal kernel data and even execute privileged instructions. In addition, buggy applications may pollute kernel memory. Hence, it is critical for UB to guarantee kernel



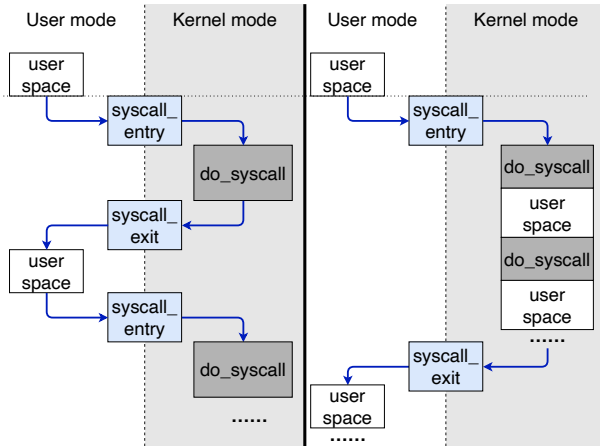


Figure 1: Invoking system calls without and with UB.

safety by performing comprehensive sanitization on userspace code and data. Finally, to achieve binary compatibility, the elevated application code should be oblivious about whether they are executed in kernel mode or user mode. The identical execution outcomes with and without UB should be guaranteed, including memory order and atomicity for multi-thread applications.

We address these challenges by adapting Dynamic Binary Translation (DBT) [52] and Software-Based Fault Isolation (SFI) [44] techniques. First, we profile syscalls by hooking their entries, to learn which syscall invocations are frequent (i.e., “hot” syscalls). Inspired by Just-In-Time (JIT) compilation [17, 22, 48], we can obtain the userspace instructions following the hot syscalls in the runtime. The instructions, if within the same function, will be translated into Binary Translation Cache (BTC). Next, we iteratively execute the BTC and extend the BTC from the exit instruction until we meet the next syscall invocation. We perform instruction and address sanitization to restrict the behaviors of BTC, and achieve kernel control-flow integrity (CFI) and data integrity on the BTC. UB does not re-order instructions or split memory access. As a result, other threads can execute concurrently and safely with the thread optimized by UB.

We implement a prototype of UB and evaluate its performance gain in I/O micro-benchmark and real-world applications including Redis and Nginx. Under our default setting (the tested application runs in a virtual machine (VM) and the Linux KPTI is turned on), I/O micro-benchmark threads can be accelerated by 30.3% to 88.3%. For Redis GET, the acceleration ratio ranges from 4.4% to 10.8% for 1B – 4KiB data sizes. Nginx can be accelerated by 0.4% – 10.9%. UB can accelerate raw socket-based packet filters by 31.5% – 34.3%. We also evaluate the impacts of KPTI and virtualization on UB’s performance gain. Since turning off KPTI reduces the syscall overhead, UB is less effective. For example, the acceleration ratio for I/O micro-benchmark drops from 88.3%

to 41.6% for the smallest I/O size. Hence, future processors, which are expected to eliminate Meltdown and Spectre vulnerability in hardware, will benefit much less from UB. When the applications run in the physical machine, UB achieves higher upper-bound acceleration ratios in most settings compared to VM, because IOPS is usually higher in this case, which results in more syscalls that can be optimized.

We also compare UB with other systems that optimize syscalls, including `io_uring` [23], F-Stack DPDK [45] and eBPF [34] in our experimental study. The results show that UB is less advantageous, comparing with `io_uring` in the micro-benchmark, F-Stack for the Redis macro-benchmark, and eBPF for raw sockets. However, UB has a unique advantage that no code change is required for the application developers.

Finally, we acknowledge UB might introduce new security risks under side-channels, undocumented x86 instructions, and kernel races. We accordingly suggest a few defense ideas.

The code of our UB prototype is published at [15]. We summarize the contributions of this paper as follows.

- We propose userspace bypass (UB), which directly executes the instructions between syscalls in kernel mode, to accelerate syscalls.
- We provide a concrete design that transparently translates userspace instructions to kernel-safe, sanitized BTC. With this method, existing applications can be executed without modification and enjoy the performance gain.
- We implement a prototype and evaluate it against several high IOPS apps. The results prove the effectiveness of UB.

## 2 Background

In this section, we first overview the syscall mechanisms and their introduced overhead. Then, we describe the prior efforts in reducing such overhead.

### 2.1 Syscalls and Their Costs

Syscall presents the default interface between userspace applications and kernel services. Software interrupt (e.g., `int 0x80`, which has been deprecated) and special instructions (e.g., `syscall/sysret` created by AMD and `sysenter/sysexit` created by Intel) can be leveraged to transfer the control from user space to kernel space and vice versa after syscall.

Previous studies have shown that syscall invocation can introduce prominent overhead on various applications and scenarios [16, 35, 43], including *direct costs* and *indirect costs* [43]. For the first case, because of switching between user mode and kernel mode, extra procedures have to be executed to save registers, change protection domains, and handle the registered exceptions. For the latter case, the state of processor structure, including L1 cache data and instruction

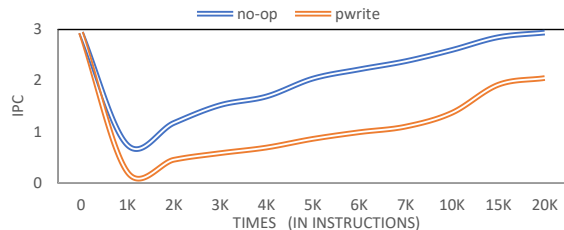


Figure 2: IPC after no-op syscall and `pwrite` syscall, measured on our platform (Intel Skylake and Linux).

caches, translation look-aside buffers (TLB), etc., can be *polluted* by syscalls, and the Out of Order Execution (OOE) of CPU has to be stalled for the order guarantee. As a result, the user-mode instructions per cycle (IPC) would be decreased after syscall.

A widely-used technique called kernel page-table isolation (KPTI) [47] makes syscalls even slower. To defeat transient execution attacks, e.g., Meltdown [29] and Spectre v3a [49], OS kernel uses two sets of page tables for user space and kernel space. As a consequence, CPU should switch to kernel page table upon entering syscall, and switch back when returning to userspace. Besides KPTI, virtualization may also increase the context-switching overhead. For example, the overhead of TLB miss (part of indirect overhead) inside VM can be larger, as more page table entries have to be examined than inside physical machines.

Below we summarize the observations from previous studies and our measurement about the concrete syscall overhead.

- A no-op system call with KPTI enabled can cost 431 CPU cycles, as measured by Mi et al. on Intel Skylake and `seL4` [35].
- As measured in our experiment platform (Intel Skylake and Linux), the kernel prologue and epilogue (direct costs) take 197 instructions (992 CPU cycles) for a no-op syscall, suggesting the issue of syscall overhead persists a decade after the study of Soares et al. [43].
- Also on our platform, a `pwrite` syscall can degrade the IPC of the following userspace instructions from 2.9 to 0.2 (indirect costs). The IPC slowly goes back to 2.1 after executing 20,000 instructions. Figure 2 shows the trend of IPC by time elapsed.

## 2.2 Performance Optimization on Syscalls

The research community is actively working on mitigating the overhead resulting from syscalls. Below we describe the related work with a comparison to our approach (also summarized in Table 1).

**Asynchronous syscalls.** Syscall introduces a synchronous execution model, as the user-mode execution is resumed after a syscall is finished. Brown proposed non-blocking Linux syscalls [5] that can be completed asynchronously parallel

Scheme	Develop Cost	Async Needed	Acceleration	Popularity
<b>eBPF</b>	++	✓	++	++
<b>DPDK</b>	++	✓	+	++
<b>io_uring</b>	++	✓	+	+
<b>UniKernel</b>	+++	-	+++	-
<b>FlexSC</b>	+	-	+	-
<b>UB</b>	-	-	+	

Table 1: Comparison of schemes for optimizing syscalls.

to the userspace execution flow. But, this approach does not completely decouple the syscall invocation from its execution. So far, most of the syscall implementations on Linux are still synchronized.

**Syscall batching.** As locality is a major performance factor, executing syscalls in a batch has also been investigated. Rajagopalan et al. proposed to group consecutive syscalls into one (the result of a syscall is directly fed to the next) [38]. This approach is effective under the assumption that no computation happens between two syscalls. Soares et al. proposed to batch syscalls of multiple co-routines and asked the developers to change the thread model to M-on-N (“M user-mode threads executing on N kernel-visible threads, with  $M \gg N$ ”) [38]. Thus, it only works when the task can be split into many threads. Modern kernel provides native queues, i.e., `io_uring` [23], to batch I/O requests from userspace processes and reduce the occurrences of syscalls. In particular, userspace code can issue multiple requests to the queue and invoke one syscall to let kernel process the queue<sup>1</sup>.

**Unikernel.** To mitigate the overhead of context switching, the Unikernel solutions run application code in kernel space instead of user space. Examples include Loadable Kernel Module (LKM) [42] and library OS [31, 40].

**In-kernel sandbox.** To reduce the occurrences of context switching caused by syscalls, in-kernel sandbox allows application code to run in privileged mode. For instance, eBPF [34] allows developers to attach code into kernel trace points. When kernel reaches these points, it will use a VM to execute the attached code. However, eBPF places many restrictions on the code, and kernel verifies if all the requirements are met before execution, during which legal codes may be rejected because of false positives in verification. Recently, Dmitry et al. propose to use in-kernel sandbox to execute applications entirely in kernel [27], in which context switching overhead can also be mitigated.

**Kernel bypass.** Observing that kernel does not always have to be involved in I/O tasks like packet handling, some researchers proposed the kernel bypass approaches. One prominent example is the Data Plane Development Kit

<sup>1</sup>`io_uring` also supports a kernel polling mode if the application has root privilege, where no syscall is required.

(DPDK) [11], which takes over I/O devices in userspace. Specifically, I/O requests are submitted to devices via a shared ring buffer, instead of syscalls. The buffer is maintained in userspace, involving no kernel activity for I/O.

We find that existing approaches all require noticeable development efforts. Different coding paradigms have to be followed in order to use the syscall-refactoring primitives. Most kernel bypass and syscall batching solutions (e.g., DPDK, RDMA, io\_uring) require application code to interact with a queue pair asynchronously. Nonetheless, developers still prefer to write program logic in the synchronous style. Refactoring the legacy code is also labor-intensive. As an example, we compare the unofficial Redis with DPDK support [2] to its official version (version 3.0.5). We find that the former includes 9,984 extra lines of code (LoC) to support DPDK, which accounts for 10% of the LoC of the official version. Another example is Unikernel: It requires developers to write kernel-mode code, which is unfortunately difficult to debug and prone to errors like memory corruption (there is no memory isolation). In addition to changing the application code, special userspace drivers may be required for kernel-bypass solutions [24].

### 3 Design Overview

To address the aforementioned issues, we propose userspace bypass (UB), a new primitive for syscall optimization. UB aims to fulfill the following three design goals (DG).

**DG1: Minimizing the manual efforts of developers.** Different from syscall-refactoring approaches, which require developers to change their legacy code or adjust to asynchronous programming, UB optimizes the syscalls at the *execution time*, which meanwhile does not impact application’s functionality.

**DG2: Minimizing changes to system architecture.** Syscall-refactoring approaches may change the current system architecture, e.g., mapping and binding devices to userspace. In contrast, UB keeps the current system architecture unchanged, including device driver and I/O harvesting models.

**DG3: Comparable performance to syscall-refactoring approaches.** UB aims to reduce the direct and indirect costs of syscalls, and achieve similar performance boost compared to syscall-refactoring approaches.

#### 3.1 Syscall-intensive Applications

We focus on optimizing applications of high IOPS, e.g., Redis and Nginx, which are also syscall-intensive. By analyzing their code and runtime behaviors, we identify the following two insights that guide the design of UB.

**Lightweight userspace instructions in I/O threads.** We find that the computation workloads between I/O events are usually lightweight for the examined applications. Moreover, the number of instructions between two consecutive syscalls

is usually small. One explanation is that such applications follow a popular I/O model that separates I/O-intensive workload from CPU-intensive workload in different threads. For example, Redis server has a main thread that dispatches accepted sockets to I/O threads [10], which conduct I/O from/to kernel and let the main thread complete the CPU intensive computation. With such a design, the instructions between I/O events simply handle buffer movement. We also profiled syscalls invoked by Redis (in total 3M), and found half of them are followed by less than 400 userspace instructions (around 200 cycles when IPC is 2) before the next syscall, which is faster than executing a syscall itself (e.g., 431 cycles [35] as described in Section 2.1).

**Amplified direct and indirect costs.** Section 2.1 overviews the direct and indirect costs of syscall in general, and those costs can be amplified in syscall-intensive applications. As shown in Figure 1, the frequency of entry and exit rises linearly following the frequency of syscall invocation. The indirect costs due to TLB misses, OOE stalls and cache misses are also non-negligible, especially when the syscall handles lighter tasks (IPC drops to 0.74 for no-op syscall, and 0.21 for `pwrite`, as shown in Figure 2).

#### 3.2 UB Modules

Based on the above considerations, we are motivated to design UB in a way that it can detect the occurrences of syscalls, and elevate the userspace instructions between consecutive syscalls to kernel through binary transformation. Figure 1 illustrates our idea. Although the idea seems simple at the high level, a few challenges should be addressed to enable UB for real-world, full-fledged applications.

- The application code is less trustworthy compared to the kernel code. Hence, necessary *isolation* should be performed to confine its capability after being moved to kernel. However, identifying the untrusted regions and governing them with the right policies are non-trivial.
- Given that isolation would incur extra costs, it is not always beneficial to transform every chunk of userspace instructions. But, when to perform transformation and how to reduce its overhead are unknown.

UB addresses these challenges with three key components. 1) A “hot” syscall identifier that monitors the execution of the target application, profiles the invoked syscalls, and determines when userspace instructions need to be elevated; 2) a *Just-in-Time (JIT)* translator that converts the userspace instructions into *Binary Translation Cache (BTC)* that is instrumented with isolation policies; 3) a *kernel BTC runtime* that executes the translated code. Figure 3 overviews the design of UB.

Note that the components in UB are not fundamentally new concepts. BTC is a standard component for Dynamic Binary Translation (DBT) [3, 22]. The JIT translator follows the guideline of Software-Based Fault Isolation (SFI) [44] in

code instrumentation and isolation policies. Yet, we find that the existing systems cannot be directly used in our problem setting. Below we briefly discuss the main modules in UB.

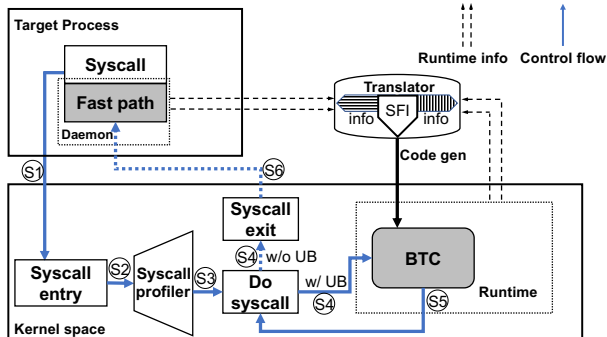


Figure 3: Overview of the UB framework. Every time a thread calls syscall (S1), hot syscall identifier hooks it (S2) and dispatches it to `do_syscall` (S3), after which kernel may return to user mode if the syscall is not hot (S6), or send it to BTC runtime for UB (S5).

**Hot syscall identifier.** This module runs in kernel mode and hooks each syscall. By analyzing the runtime statistics, it can identify which syscall instructions are *hot*, i.e., ones with high chances to be followed by another syscall shortly. The userspace instructions between two consecutive hot syscalls will be elevated to kernel and accelerated next time when the application runs. To avoid introducing large overhead due to runtime monitoring, this module runs intermittently.

**BTC translator.** The BTC translator converts the userspace instructions marked by the previous modules to BTC and has it executed by the kernel BTC runtime. Under the SFI guidelines, it converts dangerous instructions (e.g., indirect control-flow transfer) to the safe ones (e.g., direct jumps), and instrument checks to constrain memory access and control-transfer behaviors. The translator runs in a separate, independent userspace process to avoid introducing its code to kernel. The translation does not block the application execution, and the translated code is executed *next time* when the same code path is visited.

In addition to optimizing userspace instructions between a pair of hot syscalls, we also consider the acceleration on a *sequence* of hot syscalls. We call the enclosed userspace code *fast path*. UB aims to chain such userspace code and accelerate them altogether. The fast path is discovered incrementally by watching the jump targets. The details will be discussed in Section 5.

## 4 Hot Syscall Identifier

**Criteria of userspace bypassing.** A region of userspace instructions should be elevated when its performance gain out-

weighs the translation and instrumentation overhead by BTC translator. We measure the performance gain against different userspace path length (i.e., the number of instructions), and consider the regions with *short* path. The major reason is that the instrumentation costs increase rapidly for longer paths, because more instructions have to be monitored. We consider 1,000 instructions (termed  $T_{path}$ ) as the threshold for the short path length<sup>2</sup>. Through an empirical study, we observe obvious performance gain (over 20%) with this path length (see Section 6.2).

**Module design.** This module aims to discover hot syscalls that enclose a short userspace path. We resort to online analysis to achieve seamless profiling. Specifically, this module hooks syscall entry and counts the number of instructions between two consecutive syscalls. The two syscalls are classified as candidates of hot syscalls when the instruction number is less than  $T_{path}$ . Below we describe the detailed steps.

- **Syscall sampling.** Monitoring every syscall invocation will introduce high performance penalty to the application execution. Hence, we sample syscalls and conduct the follow-up analysis only when a thread is issuing syscalls frequently (e.g., I/O threads). According to our measurement on syscall-intensive applications (e.g., Redis and Nginx), at least 100K syscalls (termed  $T_{sys}$ ) are issued per second (6M per minute), and we choose to profile less than 10% of  $T_{sys}$  syscalls (up to 500K syscalls every minute). Therefore, most syscalls are not sampled and not interfered.
- **Coarse-grained profiling.** To further reduce the profiling overhead, we check whether a monitored thread invokes syscalls at high frequency. If the thread invokes less than 50K syscalls per second (half of  $T_{sys}$ ), the module will not conduct the next fine-grained syscall profiling. In this, the threads with low IOPS will be skipped.
- **Fine-grained profiling.** For a thread invoking syscalls frequently, this module further analyzes which syscall instructions are invoked frequently. The frequent ones deserve userspace bypassing as more performance boost can be gained. We monitor 15K syscalls (15% of  $T_{sys}$ ) of each round, and maintain a table recording, for each invoked syscall instruction, its location register (RIP) and a counter of how many times the next syscall is invoked within 4 microseconds (approximately the time of executing  $T_{path}$  instructions). We consider a syscall frequent when the counter is larger than 900 (6% of the profiled 15K syscalls). These syscalls and their enclosed userspace instructions will be handled by BTC translator in the next stage.

One might wonder if the performance of this module is sensitive to the parameter selection. To test the sensitivity, we

<sup>2</sup>Soares et al. consider the invocation of a syscall frequent if it is invoked once every 2,000 or less instructions [43]. We use a more conservative number to accommodate different platforms.



check if hot syscalls of Redis and Nginx, two applications used by our experiments, can be correctly discovered on three different machines: a PC with Core i5 10500 (year 2021), two servers with Xeon 8175 and 8260 (year 2017 and 2019). All hot syscalls can be correctly identified, suggesting parameter tuning could be skipped in most cases.

## 5 BTC Runtime and Translator

In this section, we describe how the BTC translator converts userspace instructions into kernel BTC and meets the security requirements. Our BTC translator follows the procedure of Dynamic Binary Translation (DBT) [19, 22, 48]. In general, given a path consisting of basic blocks in binary and triggering an event (e.g., hot syscall in our case), DBT disassembles it, translates it with a SFI rulebook, and compiles it to BTC for the future execution. Due to SFI, the malicious or unwanted behaviors of the translated code can be contained, and safely run by the BTC runtime.

### 5.1 BTC Runtime

The translated code block is executed by a BTC runtime in kernel. The BTC runtime holds local variables in kernel stack, which can be accessed by the instrumentation instructions within the BTC for policy enforcement and context switching. The local variables include: 1) the saved kernel context, i.e., callee-saved registers, 2) the values of reserved registers, and 3) the indirect jump destination information which is used to build the fast path.

Before executing the BTC, the runtime prepares the return status for userspace, i.e., through restoring the userspace context saved on syscall entry (e.g., `pt_regs` for x86\_64). After a block is finished, the runtime processes the return status of the BTC and takes further actions. The execution of a BTC might exit the runtime in the middle when the jump target is missing, e.g., when a new path is encountered. In this case, the runtime records the information about this jump and immediately returns to userspace, i.e., the jump target. We make the userspace memory accessible to the BTC runtime, so all changes on memory are kept. Changes made to registers are updated to userspace context (i.e., `pt_regs` for x86\_64), which will be written to registers when kernel returns to userspace. Therefore, userspace state changes made by the BTC are also preserved and visible to other threads, which ensures the application logic is not changed under UB.

The execution of the BTC might also exit when a syscall instruction is encountered. In this case, a fast path between two consecutive syscalls has been completely executed in kernel, which indicates a successful userspace bypass. The BTC runtime emulates the syscall trap, by looking up the syscall number against the syscall table and dispatching syscall parameters to the corresponding `do_syscall` function (i.e., executing the syscall). After `do_syscall` returns, the BTC run-

time checks if the next syscall instruction is again hot. If the answer is yes, the runtime tries to conduct another userspace bypass. In this way, `do_syscalls` and userspace bypass can be chained, which is similar to direct branch chaining of DBT. In an ideal case, *a whole thread can be executed in kernel*.

**Fast path discovery.** The performance of UB highly depends on the identification accuracy of fast path, and we leverage an incremental, JIT-style approach to achieve high accuracy. Given an entry address, i.e., the instruction next to a hot syscall, the BTC translator first discovers a part of the fast path, by disassembling the code segment of the target thread from the entry address iteratively. The potentially unreachable paths are skipped by the translator in each iteration. Specifically, the translator only follows *direct jumps* and stops at the call instructions, which forces the translator to handle code only within a function at one iteration and consider it fast path. When an indirect jump or call is indeed made later, the target information will be collected by the BTC runtime and sent to the translator to extend the fast path after replacing the jump instructions (see Section 5.2.1). Such an approach is similar to the one adopted by QEMU [9], but we do not lift the binary to its intermediate representation.

### 5.2 BTC Translator

Below we describe how the security policies are instrumented into the userspace code. We follow the SFI principles to provide *data-access policies* and *control-flow policies* [44] on *kernel*, and the implementations are inherited and extended from Nacl [52], which sandboxes the untrusted x86 native code in browser. Noticeably, Nacl assumes source code is available so SFI rules can be enforced under static compilation. In contrast, UB performs DBT on the binaries. As such, the SFI rules have to be adjusted and extended.

**Threat model.** We assume the userspace code is untrusted, which could contain *arbitrary* code and data, and the side-effects include unmediated access to kernel memory, privileged functions, etc. The goal of UB is to ensure the userspace code cannot gain more privilege (and do more harm) after it is elevated to kernel, i.e., protecting kernel's control-flow integrity. Noticeably, this goal is different from guaranteeing control-flow integrity [1] on the userspace application (elaborated in Section 5.3). We take a *conservative* approach in designing UB and avoid elevating a fast path when the consequences can not be immediately determined (e.g., the jump targets are unknown during translation). We focus on x86\_64 platform but the proposed techniques could be easily generalized to other platforms. Below we describe the implementations related to jump, register, instruction, and memory access that ensure security under this threat model.

### 5.2.1 Jump Sanitization

The inner sandbox of Nacl checks the *explicit* control flow expressed with calls and jumps, and disallows memory dereferencing on indirect jump and call instructions. The targets of jumps are confined within the sandbox. In contrast, the entire kernel memory space is open to elevated userspace code under UB. Therefore, we take different approaches to sanitize jumps.

**Direct jump.** To prevent the code in BTC from jumping to an arbitrary address, the translation only happens when the jump target is known. In other words, only direct jumps whose targets are known are processed. The address sanitization is described in Section 5.2.3.

**Indirect jump.** Yet, userspace fast path may contain indirect jumps, and we deter BTC from processing such path till the targets are known. In particular, the translator inserts checks that compare the targets against a *target address table* (similar to jumptable [22]) when encountering the associated code at first. If the target address is not in the table, the control flow will exit BTC runtime. When such an exit is triggered during executing a BTC block, the BTC runtime sends the jump instruction address (i.e., RIP of the address) and the target address to the BTC translator, and extends the fast path, as described in Section 5.1.

We show an example in Figure 4. The indirect jump (jump to RAX, located at 0x123) is initially translated to writing down the jump target (saving RAX to stack) and exiting to BTC runtime (jump to `exit_indirect_jump`). When the path P1 is firstly executed, the BTC runtime learns a target 0x456, and the information is sent to the translator, which updates the BTC by adding a target table entry. After that, the path P1 is added to the BTC, and it will not trigger `exit_indirect_jump` for the next time. If P2 is reached later, another destination 0x789 can be learnt and the BTC will be updated, so the fast path is further extended.

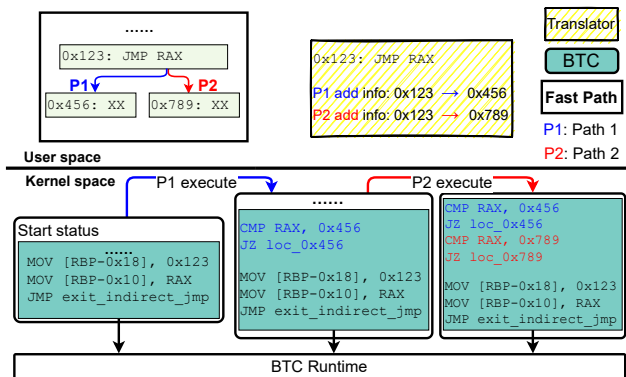


Figure 4: An example of translation under jump sanitation.

As the application runs longer, more indirect jump targets can be learned. The resulting BTC can eventually cover

the entire fast path. The checks inserted into the BTC can perform efficiently because: 1) indirect control-flow transfer instructions do not appear frequently, based on our empirical analysis on the syscall-intensive applications and previous studies [18]; and 2) CPU is allowed to speculatively jump to the destination under out-of-order execution without waiting for the destination check.

### 5.2.2 Register Remapping

To protect kernel registers and stack, the BTC translator disallows the BTC code to access stack registers (i.e., RSP, RBP, and RIP). Besides, some registers are reserved for BTC runtime and cannot be accessed by the BTC code as well. Hence we develop this module to manage the registers.

Specifically, the BTC translator uses the  $M$  reserved registers in BTC to serve the potential access to  $N$  registers ( $N = M + 3$ , 3 are for stack registers). As  $M < N$ , the translator needs to schedule registers. The  $N$  registers have their values stored in local variables, and the translator chooses one from the  $M$  reserved registers to temporally act as a special register with *renaming*. The translator also inserts code to synchronize the  $N$  registers to local variables on stack. As a result, the behaviour of the BTC code is the same as the fast path in the user space.

**Register reservation.** The translator reserves R12- R15 ( $M = 4$ ) for BTC runtime use, as they are the least frequently used in common userspace applications (less than 1% usage frequency [18]). When they appear in the fast path, renaming will occur. We also optimize our renaming mechanism for frequently-used special registers (i.e., RSP), by letting the translator fix the reserved registers to hold their values. Doing so reduces the occurrences of the costly register synchronization.

### 5.2.3 Instruction Sanitization

Privileged instructions (e.g., `sysret`) are not allowed to appear in the BTC, to avoid privilege escalation by the malicious code that exploits UB. During translation, the translator avoids elevating a fast path to the kernel if it contains any privileged instruction.

Due to register remapping, some instructions have to be rewritten. For stack operation instructions like `PUSH/POP`, the translator substitutes them with multiple instructions. Take `POP` as an example. The translator first adds an instruction to `MOV` the operand to the popped target from the memory addressed by the reserved register (i.e., the acting stack pointer), and then updates the reserved register with the new stack pointer value, i.e., plus 8.

### 5.2.4 Memory Access Sanitization

To prevent unauthorized access to the kernel memory, the translator sanitizes all memory access instructions. For ev-

ery such instruction, the translator inserts address checking instructions before the instruction, such that only userspace addresses are allowed to be accessed, i.e., the addresses start with 0. Similar to address masking of SFI [44], the translator shifts left the address by one bit and then shifts it right by one bit, to fulfill the address requirement. Two extra instructions (i.e., `SHL` and `SHR`) are introduced to this end, but our evaluation suggested the extra overhead is negligible (0.4%). Note that the added checks do not prevent BTC from accessing *unmapped* memory region and triggering page fault, and we handle it with the procedure described below.

**Page fault handling.** We modify the page fault handler to monitor the page fault events. For minor fault and major page fault, the page fault handler behaves the same for kernel mode and userspace mode. Therefore, faults caused by the userspace applications are resolved in the same way as without UB. When invalid page fault (i.e., illegally accessing some memory regions) happens, the execution of BTC code is aborted.

Nacl also isolates the memory space between the extensions and the host browser, with the help of the segmentation provided by x86 CPUs. As such, extensions' instructions can only access memory within a segment and instructions to modify segment states are not allowed. However, although `x86_64` still provides segmentation, it only adds a segment offset to the address but does not check segment boundary, which cannot be directly used for memory isolation.

### 5.3 Security Guarantees

The translated BTC has the following security properties (termed SP), and they jointly make UB fulfill SFI policies [44] on kernel.

**SP1: Kernel control-flow integrity (CFI) for BTC.** This property is guaranteed because when the BTC runtime hands control flow over to the userspace, the execution will only terminate through the exit point. More importantly, when the runtime executes the BTC, the thread cannot jump to a location unknown to the translator. For direct control-flow transfer, the destination can only be a label of a known basic block that has been translated. Indirect control-flow transfers are all translated to direct transfers by replacing the destinations. Therefore, the BTC prevents malicious code from hijacking the kernel control-flow after it is elevated.

We want to point out that UB does not claim to add extra protection against control-flow hijacking, e.g., ROP, JOP, COOP [4, 6, 8, 37, 41], and they can still occur in userspace. Though the attacker can construct gadgets when the destination checks are passed, jumping to the kernel code segment from BTC is never allowed, as it can be detected and aborted by the translator.

**SP2: Kernel data (memory and register) integrity.** For kernel context (or registers), we design BTC runtime to be compatible with the calling conventions, and the caller (kernel)

context is saved on the stack before jumping to the BTC, which is recovered before returning to kernel instructions. The context switching is lightweight, as it does not cause privilege transfer.

For kernel memory, access sanitization ensures that no kernel memory can be accessed by the sanitized instructions, hence the kernel stack will not be tainted. Though runtime local variables must be accessible by the instructions in BTC, they cannot be exploited by malicious programs to touch the kernel stack. Only intentionally inserted instructions can touch the local variables referred by the stack base pointer, which stores runtime information like swapped-out registers (see Section 5.2.2). Because kernel CFI is guaranteed, execution would never jump to these instructions.

**SP3: No privileged instructions in BTC.** It is explained in Section 5.2.3.

**SP4: Dead loop break.** We also consider the attacks and bugs against the *availability* of the system resources. For instance, userspace applications may fall into a dead loop because of bugs or intentionally. As a countermeasure, the translator maintains a counter in BTC runtime to keep track of the number of instructions already executed. Once the counter exceeds a threshold, the execution flow can exit to runtime and in turn return to userspace, which avoids the kernel being blocked by the BTC code.

### 5.4 Thread Safety

Special attention should be paid to multi-thread userspace applications, because UB has no control over other threads except the one elevated to kernel. Memory order and atomicity have to be preserved to avoid data race. Fortunately, thread safety is automatically guaranteed by the translator and we explain it below.

**Memory order.** To preserve memory order, the translator regards all userspace memory as *volatile*, and only inserts instructions between userspace instructions without optimizing the block (e.g., reordering instructions or caching memory modification in registers). Yet CPUs can still reorder memory loads and stores according to their memory model. The original memory fences placed by the userspace applications are all inherited, and the translator does not insert extra fences.

**Atomicity.** The translator takes special measures to guarantee atomicity when using multiple instructions to emulate one userspace instruction. When translating an instruction, the translator prefers to use one instruction that has the same opcode as the original one. Hence, the atomicity of the original instruction is automatically preserved. For example, instructions with a lock prefix are translated to ones still with lock (e.g., `LOCK MOV`). If more than one instruction is needed for emulation, memory load or store must be completed in a single instruction. For example, when translating `PUSH RIP`, the offset address of the next instruction must be moved to the

userspace stack top. From the view of the translator, when BTC runtime reaches the instruction, the value of `RIP` is statically known and becomes an immediate number. However, `x86_64` does not have an instruction to directly move a 64-bit intermediate value to memory. As a result, the translator generates instructions that first move the immediate value to a 64-bit reserved register and then move the 64-bit register to the top of the userspace stack.

## 6 Evaluation

We implement the prototype of UB for Linux kernel 5.4.44. The BTC runtime is implemented as a kernel module with 416 lines of C code, which hooks syscall epilogue to conduct syscall identification and manage BTC runtime. The translator is implemented with 786 lines of Python code at userspace (except the dependant Python disassembler `miasm` and gcc assembler `as`), which communicates with the BTC runtime kernel module via `sys` file. The kernel is modified by adding only 6 lines of codes to the syscall entry to allow the module to hook syscalls.

We evaluated our prototype in an I/O micro-benchmark and two real-world applications (Redis and Nginx) for macro-benchmarks. It is also compared to related technologies including DPDK, `io_uring`, and eBPF. To evaluate these applications, we set up a virtualized environment and a bare-metal environment. The bare-metal environment consists of a client machine and a server machine<sup>3</sup>, which are connected within the 40G Ethernet LAN. The virtualized environment runs on the server, with NIC pass-through being enabled. For the micro-benchmark I/O experiment, we run the tests directly on server, as it does not require network. For other scenarios, we run the client application in the client machine and the server application in the server machine, so the traffic goes through the physical network. To show the effects of virtualization and KPTI, which impact the syscall performance as explained in Section 2.1, we run each server application in four settings: KPTI on/off  $\times$  VM/physical machine. When KPTI is on, Linux turns on PCID to mitigate performance degradation. All the following tests are conducted 10 rounds, and the average IOPS or Requests Per Second (RPS) values are shown. For the results demonstrated in Section 6.1 to Section 6.4, we focus on the setting of VM with KPTI on and briefly describe how the results are changed under other settings. In Table 2, we list the acceleration ratios among different settings.

<sup>3</sup>The server machine has an Intel Xeon 8175 CPU (24 cores), 192GB memory, Samsung 980 pro NVME SSD, and Mellanox Connectx-3 NIC. It runs Ubuntu 20.04 with 5.4.44 kernel. When set up for VM, it uses QEMU-KVM 1:4.2-3, and assigns 24 cores to the VM. The client machine has an Intel Xeon 8260 CPU, 128GB memory and Mellanox Connectx-5 NIC.

	Test	VM		Physical	
w/ PTI	<b>In-mem</b>	30.3%	– 88.3%	38.4%	– 112.9%
	<b>Redis GET</b>	-3.7%	– 10.8%	-5.4%	– 6.4%
	<b>Redis SET</b>	-0.4%	– 12.4%	-3.2%	– 16.1%
	<b>Nginx</b>	0.4%	– 10.9%	-1.4%	– 13.4%
	<b>Socket</b>	31.5%	– 34.3%	30.9%	– 38.6%
w/o PTI	<b>In-mem</b>	14.3%	– 41.6%	16.4%	– 52.0%
	<b>Redis GET</b>	-2.0%	– 4.6%	-6.4%	– 3.9%
	<b>Redis SET</b>	-5.5%	– 4.9%	-0.9%	– 2.8%
	<b>Nginx</b>	-1.2%	– -0.3%	-0.2%	– 3.0%
	<b>Socket</b>	14.5%	– 17.8%	9.2%	– 19.8%

Table 2: Ranges of acceleration ratios for different settings. “In-mem” means the in-memory file access benchmark.

### 6.1 I/O Micro-benchmark

We first consider accelerating a thread that purely performs file I/O requests via blocking syscalls as the micro-benchmark, which approximates the best-case scenario for UB. The thread runs a tight loop that sequentially reads files from kernel to userspace buffer via `READ` syscall 8.39 M times. The real-world applications may exhibit different patterns like executing more instructions between consecutive I/O requests, reducing the acceleration ratios by UB. For comparison, we employ `io_uring` (liburing-2.2) for the same task (i.e., tight-loop `READ` syscall) and compare the IOPS.

**In-memory file access.** We create a large file in ramfs to avoid possible disk bottleneck, in order to assess how UB accelerates syscalls more accurately. Admittedly, this setting makes the micro-benchmark less realistic. We gradually increase the size of the buffer for each read and evaluate the acceleration ratios of UB under different buffer sizes.

Figure 5 shows the results. For the virtualized environment with KPTI on, UB accelerates syscall-based I/O by  $88.3\% \pm 0.75\%$ <sup>4</sup>, when the I/O size is small (64B). For larger I/O size, IOPS drops for both UB and baseline, and the acceleration ratio drops to  $30.3\% \pm 0.96\%$  for the 4KiB I/O size, because fewer syscalls are invoked. Turning off KPTI increases the IOPS, but the acceleration ratio of UB drops to  $14.3\% \pm 1.83\%$  –  $41.6\% \pm 1.73\%$ , because the syscall overhead is reduced. The acceleration on physical machine is higher especially when the I/O size is small (e.g.,  $112.9\% \pm 1.78\%$  when the I/O size is 64B when KPTI is on), as the IOPS on physical machine is higher and UB saves more context switching overhead.

For `io_uring`, we first examine the different queue depths (i.e., how many requests can be batched) from 1 to 1024, and found IOPS is stable after the depth reaches 128, as shown in Figure 6. Hence, we set the depth to 128 for its comparison with UB. It turns out `io_uring` yields more IOPS for most buffer sizes, according to Figure 5. When running in physical

<sup>4</sup>We report the acceleration ratio together with the standard deviation.



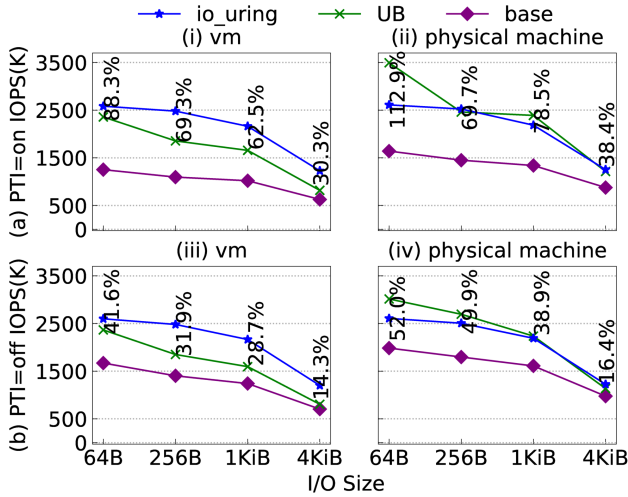


Figure 5: IOPS of READ syscalls, io\_uring and UB syscalls against different buffer sizes. The percentage number is the acceleration ratio of UB against the baseline. Figure 7, 8, 9 and 10 follow this style.

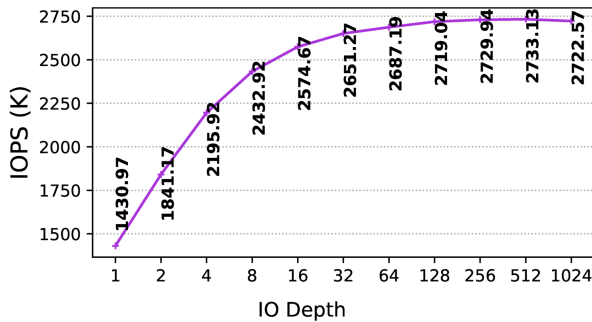


Figure 6: IOPS of different io\_uring depth.

machine, for the small size (64 Byte), UB yields higher IOPS than io\_uring, though it is expected io\_uring should always outperform UB. We have not found a good explanation, but we notice that IOPS of io\_uring increases by 13% when upgrading the Linux kernel from 5.4.44 (the version used by our testing environment) to 5.15. Hence, it is possible that io\_uring will outperform UB consistently on newer Linux.

**File access on NVMe.** We also test reading a file in NVMe disk by 1KiB block size, and show the comparison in Table 3 (“w/o sum”). We only consider the physical machine setting, because when VM accesses a file in a virtual NVMe disk, the file will be automatically cached into memory a priori, which behaves similarly to in-memory file access. The IOPS can be increased from 779 $\pm$ 5K to 852 $\pm$ 3K, yielding 9.4% $\pm$ 0.3% acceleration (KPTI on). When KPTI is off, the baseline increases to 810 $\pm$ 22K while UB increases slightly to 858 $\pm$ 10K, making the acceleration ratio smaller.

We also consider the situation that an I/O thread conducts

lightweight calculation, like parsing packets. When the computations between consecutive I/O requests have dependency, the requests cannot be batched. Specifically, we set the I/O thread to calculate the sum of the buffer by treating it as a 64-bit integer array, after retrieving the buffer from kernel.

	w/o sum	w/ sum
<b>KPTI on</b>	779 $\pm$ 5 (852 $\pm$ 3)	630 $\pm$ 4 (793 $\pm$ 3)
<b>KPTI off</b>	810 $\pm$ 22 (858 $\pm$ 10)	686 $\pm$ 65 (795 $\pm$ 6)

Table 3: KIOPS of reading file on NVMe disk (1KiB size) on physical machine (w/o sum), and reading together with integer summation (w/ sum). The UB accelerated number is shown in the bracket.

As shown in Table 3 (“w/ sum”), even the lightweight computation could reduce considerable amount of IOPS. The baseline IOPS drops by 149K, while it only drops by 59K when UB is on, as such lightweight calculations in userspace can be entirely ported into kernel for execution, so their IPC is less affected by syscalls.

## 6.2 Redis

We choose a popular key-value store engine Redis as one macro-benchmark to test how UB handles real-world workloads. We evaluate Redis 6.2.6 with the built-in Redis benchmark tool [39] to generate workload. We run the Redis server with its default configuration and launch the Redis benchmark with 2 threads. The connection number is kept at the default value 50. In each round, the client issues 1M requests.

By default, Redis completes most of its work within the main thread, which is responsible for not only I/O but also computation tasks like hashing. For a normal workflow, which is also described in [30], the main thread invokes EPOLL to get a list of readable sockets. For each readable socket, the thread READs the socket and then processes the request. As a result, the userspace paths following READs are long (from 3k to 20k), as the computation tasks happen there. At last, Redis WRITES responses to corresponding sockets one by one, with a small number of instructions in between (around 300).

**Results.** Figure 7 shows RPS with and without UB for GET and SET data of sizes ranging from 1B to 16KiB. When tested in VM with KPTI on, for GET, the acceleration ratio ranges from 4.4% $\pm$ 1.52% to 10.8% $\pm$ 2.69%, when the data size is less or equal than 4KiB. The ratio drops to -3.7% $\pm$ 0.51%, when the size rises to 16KiB. Turning KPTI off drops the acceleration ratio to between -2.0% $\pm$ 1.32% and 4.6% $\pm$ 1.96%. The negative acceleration ratio suggests the overhead brought by UB outweighs the syscall overhead saved by itself. Executing on physical machine observes a different range: -5.4% $\pm$ 1.17% to 6.4% $\pm$ 2.01% for KPTI on and -6.4% $\pm$ 3.02% to 3.9% $\pm$ 1.67% for KPTI off. Noticeably, the RPS of Redis is much smaller

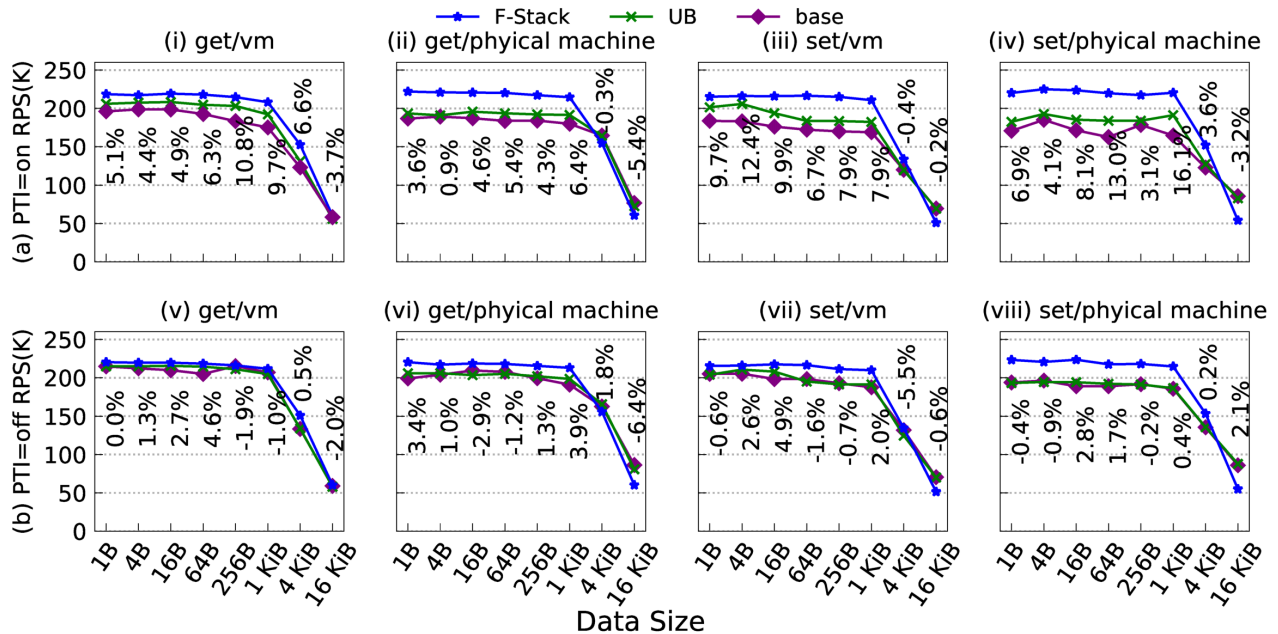


Figure 7: RPS of Redis GET and SET in different data sizes.

than that in our I/O micro-benchmark, so the expense from syscall is not the dominant factor. As a result, the acceleration ratio is much smaller.

Regarding SET, the acceleration ratio ranges from  $-0.4\% \pm 2.19\%$  to  $12.4\% \pm 3.96\%$  in VM with KPTI on. Similar trend is observed when KPTI is turned off and running in physical machine. Noticeably, Redis RPS drops significantly over 1KiB data size for both SET and GET, and similar observation was reported in the official documentation of the Redis benchmark [39].

Surprisingly, we found the RPS on VM is often higher than physical machine, though the virtual setup is supposed to yield lower RPS. We do not have a good explanation for why the opposite happened for Redis.

	Redis Server	BTC	User Space	Do Write
w/o UB	108.57	-	34.29	36.73
w/ UB	102.98	2.42	28.38	36.07

Table 4: Time spent on each part of Redis (VM+KPTI, SET).

**Profiling performance gain.** We let the BTC runtime profile BTC execution and userspace execution using the RDTSCP instruction. We run 20M Redis SET transactions (about 100 seconds) with and without UB. Table 4 shows the results. As we can see, by elevating the fast path to BTC, 5.91s userspace time can be saved, while the BTC only costs 2.42s. The difference (3.49s) can be attributed to the userspace IPC increase (indirect overhead). 5.59s are saved in total (the “Redis Server” column) and 2.1s (i.e., 5.59s - 3.49s) are saved directly by invoking fewer syscalls.

**Overhead of memory checks.** When strong kernel memory safety is unnecessary, e.g., when the binary is formally verified, a user may choose to chase higher performance gain by removing the instructions inserted to check memory boundary (i.e., SHL and SHR). We evaluate how much RPS gain can we get if we ask the translator not to insert such instructions. The results show that only 0.4% more RPS can be gained.

**Comparison with DDPK.** We compare the acceleration ratio of UB on Redis with that on DDPK as there are open-source implementations to empower Redis, like Redis-DDPK [2] and F-Stack Redis [45]. We chose F-Stack as the maintenance of Redis-DDPK has stopped since 2017 and it cannot run on the latest CPUs. F-stack supports the recent Redis 6.2.6 [46] as well as the recent DDPK 20.11. The comparison result is also shown in Figure 7.

It turns out F-Stack provides higher acceleration ratios for small size consistently (no larger than 4KiB). Interestingly, we found for 16KiB, F-Stack performs worse than UB and Redis baseline. One potential explanation is that F-Stack does not benefit from our multi-core setting. When we measure the CPU usage, it is always 100% for F-Stack, but UB and baseline can go up to 124%, which means multiple cores are used. Hence, F-Stack might outperform UB consistently when we restrict the core number to 1.

### 6.3 Nginx

In addition to Redis, we use Nginx (Version 1.20.0), a popular static web server with high RPS, as another macro-benchmark. Table 5 shows the number of instructions in the path followed by each syscall. These followed by less than 1,000 instruc-

syscall	recvfrom	openat	fstat	setsockopt	writew	sendfile	close	setsockopt
#Instructions followed	4,328	38	4,412	43	177	541	477	509

Table 5: Number of instructions following each syscall of Nginx. Those followed by less than 1,000 instructions are hot. The two setsockopt calls are different.

tions can be regarded as hot. Therefore, 6 out of the 8 can be accelerated. We run wrk [50] (Version 4.1.0, with 8 threads and 1024 connections), an HTTP benchmark tool, on the client machine to issue requests to the Nginx server for 12s to examine how much RPS Nginx can handle.

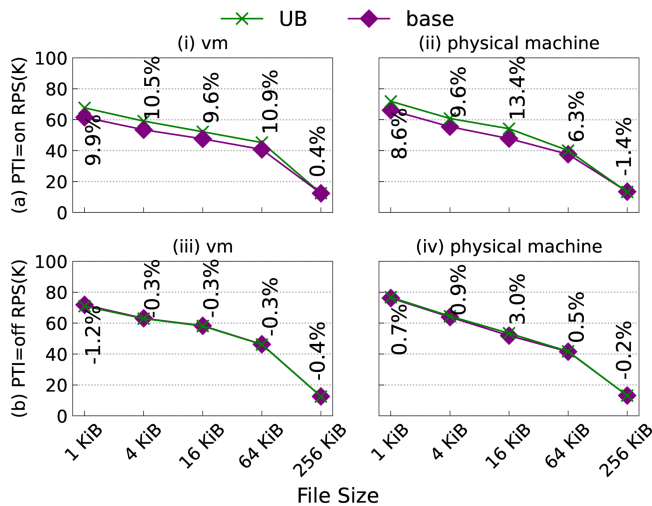


Figure 8: RPS of Nginx against different file sizes (Bytes).

**Results.** We gradually increase the file size requested by wrk and Figure 8 shows the RPS before and after UB acceleration. When being tested in VM with KPTI on, Nginx can be accelerated by  $9.6\%_{\pm 1.81\%}$  to  $10.9\%_{\pm 0.22\%}$  for 1KB to 64KB files, but the ratio drops to  $0.4\%_{\pm 0.86\%}$  for 256KB file. For physical machine, the acceleration ratio ranges from  $6.3\%_{\pm 0.17\%}$  to  $13.4\%_{\pm 3.32\%}$  for 1KB to 64KB files, but also drops to  $-1.4\%_{\pm 0.28\%}$  for 256KB file. These results show the bottleneck shifts from syscall to I/O for large files. When turning off KPTI, UB does not yield noticeable acceleration.

**Multiple worker threads.** We evaluate how multi-threading affects the acceleration ratio. We gradually increase the number of worker threads of Nginx and evaluate the case of 4KB file size. Figure 9 shows the RPS. As we can see, with more worker threads, the acceleration ratio drops noticeably when KPTI is on (from  $8.6\%_{\pm 0.22\%}$  to  $7.1\%_{\pm 0.17\%}$  for VM and  $4.7\%_{\pm 0.15\%}$  to  $2.0\%_{\pm 0.26\%}$  for physical machine), as the worker threads are increased from 2 to 8. When there are more worker threads, more cycles are used for thread synchronization, so fewer requests can be served per thread, reducing the syscall overhead saved by UB.

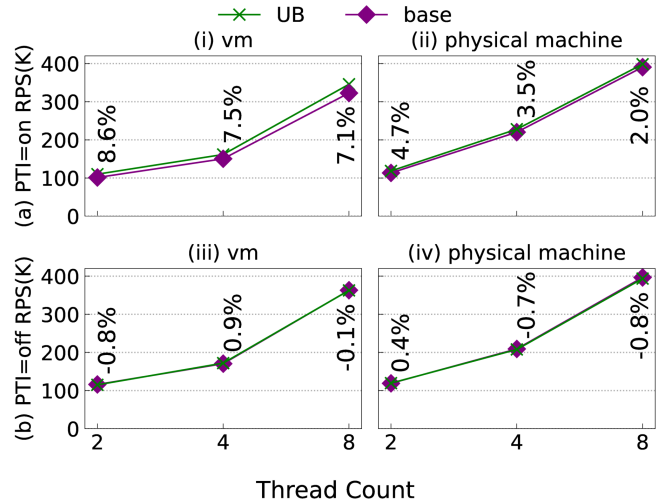


Figure 9: RPS of Nginx against different # of threads.

## 6.4 Raw Socket vs. eBPF

To avoid the syscall overhead, eBPF is another popular solution as described in Section 2.2. We show that, with the help of UB, developers can simply write the processing logic entirely in userspace with *raw socket*, and compare Packets Per Second (PPS) with eBPF.

We run a program on the client machine to send UDP packets to the server, and the server handles the incoming packets by either raw socket or XDP (eBPF library for packet processing) for 12s in each round. The client runs 15 threads, which can saturate the server. The processing tasks include counting the number of packets and summing packets by treating a packet as an integer array.

**Results.** Figure 10 shows the results by 3 packet sizes (128B, 512B, and 1472B). For VM with KPTI on, eBPF outperforms raw socket for small packets by up to  $368.4\%_{\pm 8.92\%}$ . For packets of MTU size (i.e., 1472B), eBPF still has  $236.7\%_{\pm 4.15\%}$  more PPS. UB accelerates raw socket by  $31.5\%_{\pm 0.25\%}$  –  $34.3\%_{\pm 0.72\%}$ , which are much smaller than eBPF. The PPSs for raw socket are similar across different packet sizes. However, eBPF is very sensitive to packet size, and we believe it is because the bottleneck of raw socket is protocol stack processing, which is bypassed by eBPF whose bottleneck may be the data movement, whose time consumption is related to packet size. When KPTI is off, the acceleration ratio of UB drops to  $14.5\%_{\pm 0.45\%}$  –  $17.8\%_{\pm 0.44\%}$  for various packet sizes. On physical machine, the acceleration ratios of UB

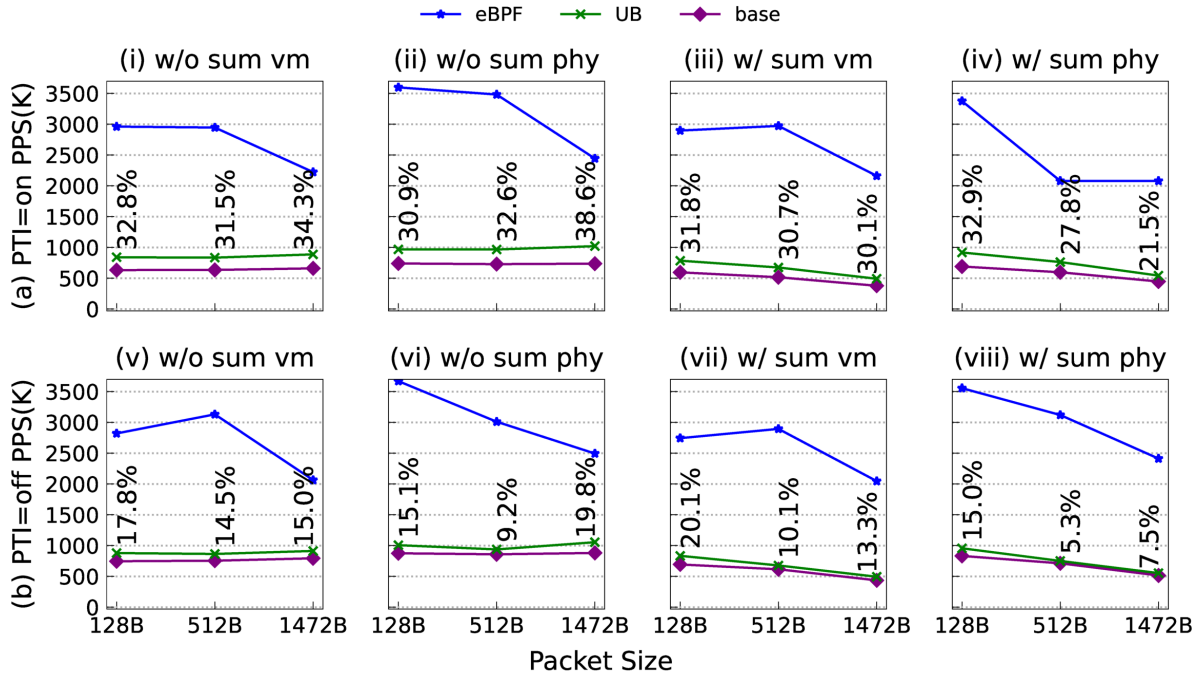


Figure 10: PPS of server handling incoming UDP packets in different packet sizes.

have larger ranges ( $30.9\%_{\pm 0.87\%} - 38.6\%_{\pm 0.56\%}$  for KPTI on and  $9.2\%_{\pm 0.14\%} - 19.8\%_{\pm 0.31\%}$  for KPTI off).

**Computation.** We also consider adding lightweight computation workload, i.e., packet summing, like the experiments for NVMe file access (Section 6.1). In VM, the PPS of raw socket sees greater drop when the packet size increases, but UB can still accelerates raw socket in similar ratios ( $30.1\%_{\pm 0.20\%} - 31.8\%_{\pm 0.50\%}$  for KPTI on and  $10.1\%_{\pm 0.18\%} - 20.1\%_{\pm 0.15\%}$  for KPTI off). eBPF is able to keep the similar PPS without packet summing. On the physical machine, similar trend is observed for raw socket, UB, and eBPF, except that eBPF sees considerable drop of PPS for 512B packet size and KPTI on.

**Profiling execution performance.** We profile the execution time of BTC and eBPF respectively for the case of packet summing with `RDTSCP`, like our experiments on Redis (Section 6.2). In VM with KPTI on, for handling 33.85M incoming packets of 128B, BTC spent 5.86s. In contrast, eBPF costs 9s. As we can see, the execution performance of BTC is better than eBPF VM. However, UB still cannot achieve similar PPS to eBPF based on the previous results. According to our analysis, the reason is that eBPF runs in `softirq`, so the packets can be dispatched into different cores. In contrast, the raw socket protocol stack has in-kernel locks for concurrent access. In particular, we added more threads for socket read, but did not see PPS increase at all. We also tried to assess how eBPF works without multi-threading, by restricting the IRQ of the NIC to a single core and repeating the sum experiment in VM with KPTI on. UB-accelerated socket reaches 1M, 0.96M and

0.93M PPS for the three packet sizes, while eBPF reaches 0.96M, 0.93M and 0.91M PPS respectively. Therefore, we believe the PPS of raw socket can be significantly improved if kernel optimizes its protocol stack for concurrent access. One potential approach is to build a better UB runtime so more deeper kernel trace points can be exposed via syscall, and we leave this as a future work.

## 7 Discussion

### 7.1 UB vs. eBPF

In addition to the comparison on the performance between UB and eBPF, here we compare their restrictions and security guarantees. As eBPF is developed mostly for packet processing and kernel tracing, it has a number of restrictions on the application code. For example, eBPF is not Turing Complete, as infinite loops are not allowed [33]. Due to its extensive restrictions on code, the eBPF verifier is prone to produce false positives, i.e., legal code regarded as illegal [14]. UB does not add any restrictions to developers and translates the userspace code transparently.

Regarding performance, UB only accelerates the paths following syscalls, but eBPF can be attached to many tracing points inside kernel, which makes it more flexible and capable of overcoming kernel bottlenecks. We believe UB could realize similar performance as eBPF, if kernel exposes more tracing points via syscalls.



In terms of security, eBPF relies on the isolation from in-kernel VM, while UB relies on the policies of SFI translator. Attacks targeting eBPF might be effective against UB as well, as described in Section 7.2. Formally verifying the implementation of eBPF and UB could mitigate these issues, but verifying eBPF is likely easier than UB, because eBPF has an official specification and it uses a reduced set of instructions.

## 7.2 Security Risks

Though we follow the SFI principles to design UB, new security risks could be introduced. First, UB might be vulnerable under side-channel attacks, which infer the secrets according to micro-architectural state changes. For instance, the Spectre attack has demonstrated that eBPF can be exploited to steal kernel memory, as eBPF VM compiles userspace code into kernel code [25]. The BTC of UB may also be exploited for similar attacks. To mitigate such risk, defenses against speculation attacks should be considered, e.g., placing speculation blocking instructions by the compiler [25]. Second, our BTC translator might not be able to sanitize privileged undocumented X86 instructions. To mitigate the introduced risk, the translator could allow a whitelist of instructions. When instructions outside the whitelist are encountered, UB should give up elevating their fast path. Third, previous research showed kernel races can lead to time-of-check to time-of-use (TOCTOU) attacks [28]. Since the BTC runtime does not enforce atomicity between the checking point and the use point for the fast path, the malicious userspace code can exploit kernel races. The mitigation can rely on the existing defenses that detect kernel races actively [20].

## 7.3 Other Limitations

Admittedly, kernel-bypass frameworks like DPDK could achieve better performance than UB, when the developers take the right measures to integrate them into the userspace applications. The better performance not only comes from the reduction of context switching overhead, but also the simplified and more efficient userspace drivers. For example, userspace drivers could avoid unnecessary buffer copying, interrupt, etc. In contrast, UB only reduces the context switching overhead. The key advantage of UB is that it does not require any change on the applications by the developers (see Table 1). Therefore, we believe kernel bypass would be favored when the developers are willing to refactor their code or design a new application with kernel bypass in mind.

UB does not aim to replace asynchronous I/O. Admittedly, when an application is both computation-intensive and I/O intensive, asynchronous I/O helps the developers decouple I/O from computation in different threads, making better use of multi-cores. UB does not give synchronous I/O tasks more IOPS than asynchronous tasks, but it can be used jointly with asynchronous I/O. In some cases, the I/O threads of asyn-

chronous tasks still intensively invoke syscalls to submit I/O and UB can accelerate these tasks.

## 8 Related Work

Section 2 has surveyed related works about syscall optimization. Below we describe other related works.

**Dynmaic Binary Translation (DBT).** DBT is a powerful method for debugging and instrumentation [3, 19, 22, 48]. Kedia et al. proposed a fast DBT in kernel to instrument kernel code [22]. Our translator has some similarities with theirs in indirect branch processing, but our translator differs largely in memory protection and register renaming. Besides, some functionalities of their runtime require rollback. In contrast, our runtime never rolls back.

**Software-Based Fault Isolation (SFI).** Enforcing SFI in kernel is not an entirely new idea. XFI was firstly proposed to isolate kernel modules with SFI, and later LXFI added kernel API check to restrict the fault propagated via kernel APIs [13, 32]. UB uses SFI in a different way for the fast path.

**Accelerating Inter-Process Communication (IPC).** Some schemes were proposed recently to exploit hardware assistance to accelerate IPC. Similar to accelerating system calls, they also try to minimize context switching overhead. Gu et al. proposes to accelerate IPC with the help of recent innovation in Intel processors, i.e., MPK [16]. Mi et al. borrows a hardware function designed for virtualization to accelerate IPC [35]. Du et al. proposes to add new features to CPU for context switching without involving kernel [12]. They implemented the prototype on RISC-V FPGA processors.

## 9 Conclusion

The overhead brought by syscalls is prominent to high-IOPS applications, but the existing approaches have not completely addressed this issue, because they require efforts in code refactoring. To preserve binary compatibility, we propose userspace bypass (UB) that executes userspace instructions directly in kernel. UB employs a JIT translator that translates userspace instructions between syscalls into sanitized code blocks. The code blocks are constrained to avoid introducing extra harm, therefore they can be executed directly in kernel. With UB, I/O micro-benchmark can be accelerated by 30.3 – 88.3% and real-world applications like Redis can be accelerated by 4.4 – 10.8% for 1B – 4KiB data sizes under GET, when the applications are executed in VM with KPTI on.

## Acknowledgement

We thank our shepherd Dan Tsafir for his highly valuable suggestions. The Fudan authors are sponsored by National Key R&D Program of China (Grant No. 2022YFB3102901) and Natural Science Foundation of Shanghai (No. 23ZR1407100).

## References

- [1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.
- [2] ansyun. DPDK-Redis. <https://github.com/ansyun/dpdk-redis>. Accessed: 2021-05-05.
- [3] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.
- [4] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.
- [5] Zach Brown. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 81–85, 2007.
- [6] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.
- [7] Jeff Caruso. 1 million IOPS demonstrated. <https://www.networkworld.com/article/2244085/1-million-iops-demonstrated.html>. Accessed: 2021-12-01.
- [8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, page 559–572, New York, NY, USA, 2010. Association for Computing Machinery.
- [9] Vitaly Chipounov and George Candea. Dynamically translating x86 to LLVM using QEMU. Technical report, EPFL, 2010.
- [10] Alibaba Cloud. Improving Redis performance through multi-thread processing. <https://alibaba-cloud.medium.com/improving-redis-performance-through-multi-thread-processing-ca4d8353523f>. Accessed: 2020-11-30.
- [11] DPDK. Data Plane Development Kit. <https://www.dpdk.org/>. Accessed: 2021-05-01.
- [12] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 671–684, 2019.
- [13] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.
- [14] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.
- [15] GlareR. Code repository of this project. <https://github.com/GlareR/UserspaceBypass>. Accessed: 2022-09-25.
- [16] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, pages 401–417, 2020.
- [17] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 104–113, 2012.
- [18] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. An analysis of x86-64 instruction set for optimization of system softwares. *Planning perspectives*, page 152, 2011.
- [19] Andrew Jeffery. Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU. *University of Adelaide Honors Thesis*, 2009.
- [20] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Rizzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.
- [21] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

- [22] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 101–115, 2013.
- [23] Kernel.dk. Efficient IO with io\_uring. [https://kernel.dk/io\\_uring.pdf](https://kernel.dk/io_uring.pdf). Accessed: 2021-12-01.
- [24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.
- [25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.
- [26] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
- [27] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.
- [28] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.
- [29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.
- [30] Zhiyun Luo. How does Redis process requests? (translated). <https://www.luozhiyun.com/archives/674>. Accessed: 2022-09-25.
- [31] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.
- [32] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nikolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.
- [33] Andrea Mayer, Pierpaolo Loreti, Lorenzo Bracciale, Paolo Lungaroni, Stefano Salsano, and Clarence Filisfilis. Performance monitoring with H<sup>2</sup>: Hybrid kernel/eBPF data plane for SRv6 based hybrid SDN. *Computer Networks*, 185:107705, 2021.
- [34] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.
- [35] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.
- [36] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.
- [37] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security and Privacy*, 10(6):84–87, 2012.
- [38] Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, and Richard D Schlichting. Cassyopia: Compiler assisted system optimization. In *HotOS*, volume 3, pages 1–5, 2003.
- [39] Redis. Redis Benchmark. <https://redis.io/docs/reference/optimization/benchmarks/>. Accessed: 2022-09-25.
- [40] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: a library OS with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 546–558, 2021.
- [41] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.
- [42] Amol Shukla, Lily Li, Anand Subramanian, Paul AS Ward, and Tim Brecht. Evaluating the performance of user-space and kernel-space web servers. In *CASCON*, volume 4, pages 189–201, 2004.

- [43] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association.
- [44] Gang Tan. *Principles and implementation techniques of software-based fault isolation*. Now Publishers, 2017.
- [45] Tencent. F-Stack. <https://github.com/F-Stack/f-stack>. Accessed: 2022-09-25.
- [46] Tencent. F-Stack Redis. <https://github.com/F-Stack/f-stack/tree/dev/app/redis-6.2.6>. Accessed: 2022-09-25.
- [47] The kernel development community. Page table isolation (PTI). <https://www.kernel.org/doc/html/latest/x86/pti.html>. Accessed: 2021-12-01.
- [48] Nigel Topham and Daniel Jones. High speed CPU simulation using JIT binary translation. In *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, 2007.
- [49] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.
- [50] wg. wrk. <https://github.com/wg/wrk>. Accessed: 2020-12-15.
- [51] Ziyue Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.
- [52] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.
- [53] Kai Yu, Chengfei Zhang, and Yunxiang Zhao. Web service appliance based on unikernel. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 280–282. IEEE, 2017.

## A Artifact Appendix

### Abstract

Our artifact includes the source code of UB, and the apps we used for evaluation. The readers can follow the instructions to modify the Linux kernel to support UB, compile UB to run on it, and evaluate the apps on it.

### Scope

The IOPS of all the apps we evaluated can be reproduced. Specifically, Figure 5, Figure 7, 8, 9 and 10. Reproducing the I/O benchmark is the most convenient case. Therefore, it is recommended to start from Figure 5.

The whole experiment can be time-consuming, so people may take fewer repeat rounds to save time.

### Content

The artifact includes the implementation of UB, which consists of the three files to be modified over Linux Kernel (zz\_lkm, zz\_daemon, and zz\_disassem). zz\_lkm is the kernel part of UB, which profiles processes and executes the BTC. zz\_daemon sits at userspace to communicate with the kernel module and invoke zz\_disassem to do the actual translation.

### Hosting

The source codes are hosted at <https://github.com/glaerer/UserspaceBypass>, as well as the readme file.

### Requirement

The I/O benchmark experiment requires only a server machine. Because Redis, Nginx, and raw socket experiments involve network, another client machine is required to be connected to the server.

The IOPS is highly related to CPU performance. Therefore, the reproduced IOPS values may be different by different CPUs, but we can always see the performance gain.

The IOPS can also be disturbed by network performance. If the NIC used is not sufficiently powerful, the IOPS may drop for large I/O size, as well as the performance gain.





# Triangulating Python Performance Issues with SCALENE

Emery D. Berger

*College of Information and Computer Sciences  
University of Massachusetts Amherst  
emery@cs.umass.edu*

Sam Stern

*College of Information and Computer Sciences  
University of Massachusetts Amherst  
jstern@cs.umass.edu*

Juan Altmayer Pizzorno

*College of Information and Computer Sciences  
University of Massachusetts Amherst  
jpizzorno@cs.umass.edu*

## Abstract

This paper proposes the SCALENE Python profiler. SCALENE precisely and simultaneously profiles CPU, memory, and GPU usage, all with low overhead. SCALENE’s CPU and memory profilers help Python programmers direct their optimization efforts by distinguishing between inefficient Python and efficient native execution time and memory usage. SCALENE’s memory profiler employs a novel sampling algorithm that lets it operate with low overhead yet high precision. It also incorporates a novel algorithm that automatically pinpoints memory leaks within Python or across the Python/native boundary. SCALENE tracks a new metric called copy volume, which highlights costly copying operations that can occur when Python silently converts between native and Python data representations, or between CPU and GPU. Since its introduction, SCALENE has been widely adopted, with over 675,000 downloads to date. We present experience reports from developers who used SCALENE to achieve significant performance improvements and memory savings.

## 1 Introduction

Python is now firmly established as one of the most popular programming languages, with first place rankings from TIOBE [42] and IEEE Spectrum [6], second place on the Redmonk Rankings [24], and fourth place in the 2022 Stack Overflow Developer Survey [38]. Large-scale industrial users of Python include Dropbox [4], Facebook [18], Instagram [15], Netflix [22], Spotify [47], and YouTube [44].

At the same time, Python is (in)famously slow. The standard Python implementation, known as CPython, is a stack-based bytecode interpreter written in C [48]. Pure Python code typically runs 1–2 orders of magnitude slower than native code. As an extreme example, the Python implementation of matrix-matrix multiplication takes more than 60,000× as long as the native BLAS version.

Python’s performance costs are nearly matched by its high memory overhead. Python data types consume dramatically

more memory than their native counterparts. For example, the integer 1 consumes 4 bytes in C, but 28 bytes in Python; “a” consumes 2 bytes in C, but 50 bytes in Python. This increased space demand is primarily due to metadata that Python maintains for every object, including reference counts and dynamic type information. Python is a garbage collected language; because garbage collection delays memory reclamation, it can further increase the amount of memory consumed compared to native code [14].

Because of these costs, one of the most effective ways for Python programmers to optimize their code is to identify performance-critical and/or memory-intensive code that uses pure Python, and replace it with native libraries. Python’s ecosystem includes numerous high-performance packages with native implementations, which are arguably the key enabler of its adoption and popularity. These libraries include the NumPy numeric library [25], the machine learning libraries SciKit-Learn [29] and TensorFlow [2, 3], among many others. By writing code that makes effective use of these packages, Python programmers sidestep Python’s space and time costs, and at the same time take full advantage of hardware resources like multiple cores, vector instructions, and GPUs.

Unfortunately, past Python profilers—which can be viewed as ports of traditional profilers for native code—fall short. We believe Python programmers need a profiler designed from the ground up to meet the specific challenges of developing high performance Python applications.

This paper proposes SCALENE, a profiler that comprises a suite of profiling innovations designed specifically for Python. Unlike all past Python profilers, SCALENE simultaneously profiles CPU, memory usage, and GPU usage. It provides fine-grained information targeted specifically at the problems of optimizing Python code. In particular, SCALENE teases apart time and memory consumption that stem from Python vs. native code, revealing where programmers can optimize by switching to native code. SCALENE reports a new metric, *copy volume*, that helps identify costly (and often inadvertent) copying across the Python/native divide, or copying between CPU and GPU. Its memory profiler accurately tracks memory

Profiler	Slowdown	Lines or Functions	Unmodified Code	Threads	Multi-processing	Python vs. C Time	System Time	Profiles Memory	Python vs. C Memory	GPU	Memory Trends	Copy Volume	Detects Leaks
<i>CPU-only profilers</i>													
pprofile (stat.)	1.0×	lines	✓	✓	-	-	-	-	-	-	-	-	-
py-spy	1.0×	lines	✓	✓	✓	-	-	-	-	-	-	-	-
pyinstrument	1.7×	functions	✓	-	-	-	-	-	-	-	-	-	-
cProfile	1.7×	functions	✓	-	-	-	-	-	-	-	-	-	-
yappi wallclock	3.2×	functions	✓	✓	-	-	-	-	-	-	-	-	-
yappi CPU	3.6×	functions	✓	✓	-	-	-	-	-	-	-	-	-
line_profiler	2.2×	lines	-	-	-	-	-	-	-	-	-	-	-
Profile	15.1×	functions	✓	-	-	-	-	-	-	-	-	-	-
pprofile (det.)	36.8×	lines	✓	✓	-	-	-	-	-	-	-	-	-
<i>memory-only profilers</i>													
fil	2.7×	lines	-	-	-	-	-	peak only	-	-	-	-	-
memory_profiler	≥37.1×	lines	-	-	-	-	-	RSS	-	-	-	-	-
memray	4.0×	lines	-	✓	-	-	-	peak only	✓	-	-	-	-
<i>CPU+memory profilers</i>													
Austin (CPU+mem)	1.0×	lines	✓	✓	✓	-	-	RSS	-	-	-	-	-
Scalene (CPU+GPU)	1.0×	both	✓	✓	✓	✓	✓	-	-	✓	-	-	-
Scalene (all)	1.3×	both	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

**Figure 1: SCALENE vs. past Python profilers.** SCALENE provides vastly more information than past Python profilers, with more accurate memory profiling (§6.3) and with low overhead (§6.4, §6.5). Most past profilers (§8.5) exclusively profile either CPU time or memory; SCALENE simultaneously profiles CPU, GPU, and (optionally) memory, and comprises a suite of unique features backed by novel algorithms.

consumption over time, and automatically identifies memory leaks, whether within Python or in native code. Its GPU profiler tracks GPU utilization and memory consumption, letting it identify when native libraries are not being used to their best advantage. At the same time, SCALENE imposes low overhead (median: 0% for CPU+GPU, 32% for CPU+GPU+memory). SCALENE’s design addresses the substantial differences between Python and past environments such as JVMs, including the widespread use of native libraries, the resulting reduced usage of garbage collection, and its popularity for machine learning applications that make GPUs a first-class concern.

Since its introduction, SCALENE has become a popular tool among Python developers, with over 600,000 downloads to date. We report on case studies supplied by external users of SCALENE, including professional Python open source developers and industrial users, highlighting how SCALENE helped them diagnose and then remedy their performance problems, leading to improvements ranging from 45% to 125×.

This paper makes the following contributions: it proposes SCALENE, a profiler specifically tailored to Python; it presents several novel algorithms, including its algorithm for attributing time consumption to Python or native code; its sampling-based memory profiling that is both accurate and low overhead; and its automatic memory leak detector, which identifies leaks with low overhead. It also introduces and demonstrates the value of a new metric, *copy volume*, that surfaces hidden costs due to copying.

The next sections explain SCALENE’s implementation and algorithms. We first outline how SCALENE efficiently performs line-level CPU profiling, focusing on its approach to

teasing apart time spent running in the Python interpreter from native code execution and system time (§2). We then describe SCALENE’s memory profiling component (§3), including its *threshold-based* sampling approach that reduces overhead while ensuring accuracy, its memory leak detection algorithm, and how it tracks copy volume. We then explain how SCALENE profiles GPU utilization and memory consumption (§4). Finally, we present technical details underpinning SCALENE’s user interface (§5). We then present our evaluation (§6) and a number of case studies of user experiences with SCALENE (§7); we conclude with a discussion of related work (§8).

## 2 CPU Profiling

SCALENE’s CPU profiler employs sampling, but unlike past profilers, it leverages an apparent limitation of how Python delivers signals to extract more granular information.

Sampling profilers like SCALENE work by periodically interrupting program execution and examining the current program counter. Given a sufficiently large number of samples, the number of samples each program counter receives is proportional to the amount of time that the program was executing. Sampling can be triggered by the passage of real (wall-clock) time, which accounts for CPU time as well as time spent waiting for I/O or other events, or virtual time (the time the application was scheduled for execution), which only accounts for CPU time.

However, in Python, using sampling to drive profiling leads to erroneous profiles. Like other scripting languages such as Perl and Ruby, Python only delivers signals to the main

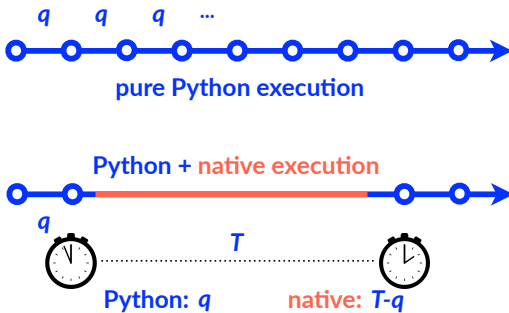


hover over bars to see breakdowns; click on [COLUMN HEADERS](#) to sort.

pytorch-mnist.py: % of time = 65.5% out of 256.1s.

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY (MB/s)	GPU util.	GPU memory	LINE PROFILE (click to reset order)
					10			39 for batch_idx, (data, target) in enumer
					4			59 for data, target in test_loader:
								44 loss.backward()
								45 optimizer.step()
					5			131 model = Net().to(device)
					5			22 x = self.conv1(x)
								124 dataset1 = datasets.MNIST('./data', tr

**Figure 2: An example profile from SCALENE’s web UI**, sorted in descending order by GPU utilization. The top graphs provide a summary for the entire program, with more detailed data reported for each active line (and, not shown, for each function). CPU time is in blue, with different shades reflecting time taken in Python code, native code, or system/GPU time (§2). Average and peak memory consumption is in green, with different shades distinguishing memory consumed by Python objects vs. native ones (§3); the memory timeline depicts memory consumption over time (§5). Copy volume is in yellow (§3.5), as well as GPU utilization and GPU memory consumption (§4). Hovering over bars provides detailed statistics in hoverslips.



**Figure 3: Overview of SCALENE’s inference of Python vs. native execution.** Sampling profilers depend on regular timer interrupts, but Python defers all signals when running native code, leading to the appearance of no time spent executing that code. SCALENE leverages this apparent limitation to accurately attribute time spent executing Python and native code (§2.1) in the main thread; it uses a different algorithm for code running in threads (§2.2).

thread [31]. Also like those languages, Python defers signal delivery until the virtual machine (i.e., the interpreter loop) regains control, and only checks for pending signals after specific opcodes such as jumps.

The result is that, during the entire time that Python spends executing external library calls, no timer signals are delivered. The effect can be that the profiler will reflect *no time* spent executing native code, no matter how long it actually took. In addition, because only main threads are interrupted, sampling profilers can fail to account for any time spent in child threads.

## 2.1 Accurate Python-Native Code Profiling

SCALENE’s CPU profiler turns these limitations of Python signals to its advantage, inferring whether a line spent its time executing Python or native code (e.g., C). It leverages the following insight: *any delay in signal delivery corresponds to time spent executing outside the interpreter*. That is, if SCALENE’s signal handler received the signal immediately (that is, in the requested timing interval), then all that time must have been spent in the interpreter. If it was delayed, it must be due to running code outside the interpreter, which is the only cause of delays (at least, in virtual time).

Figure 3 depicts how SCALENE handles signals and attributes time to either Python or native code. SCALENE tracks time between interrupts recording the current virtual time whenever it receives a CPU timer interrupt (using `time.process_time()`). When it receives the next interrupt, it computes  $T$ , the elapsed virtual time, and compares it to the timing interval  $q$  (for quantum).

SCALENE uses these values to attribute time spent to Python or native code. Whenever SCALENE receives a signal, SCALENE walks the Python stack until it reaches code being profiled (that is, outside of libraries or the Python interpreter itself), and attributes time to the identified line of code. SCALENE maintains two counters for every line of code being profiled: one for Python, and one for native code. Each time a line is interrupted by a signal, SCALENE increments the Python counter by  $q$ , the timing interval, and it increments the native counter by  $T - q$ , the delay.



## 2.2 Accurate Python-Native Profiling of Threads

The approach described above attributes execution time for Python vs. native code in the main thread, but it does not attribute execution time at all for subthreads, which, as described above, never receive signals. To correctly attribute time for code running in subthreads, SCALENE applies an algorithm leveraging a combination of Python features: *monkey patching*, *thread enumeration*, *stack inspection*, and *bytecode disassembly*.

Monkey patching refers to the redefinition of functions at runtime. SCALENE uses monkey patching to ensure that signals are always received by the main thread, even when the main thread is blocking (e.g., waiting to join with child threads). SCALENE replaces blocking functions like `threading.join` with variants that always use timeouts. It sets these timeouts to Python’s own thread quantum, obtained via `sys.getswitchinterval()`. Replacing these calls forces the main thread to yield regularly and allow signal delivery.

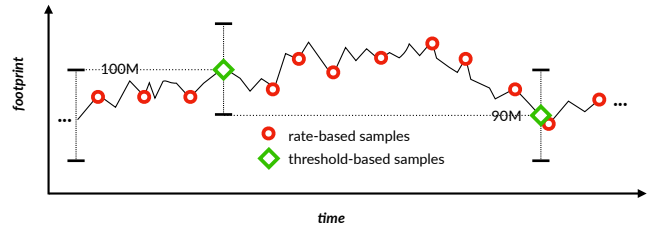
In addition, to attribute execution times correctly, SCALENE maintains a status flag for every thread, all initially *executing*. In each of the calls it intercepts, before SCALENE issues the blocking call, it sets the calling thread’s status as *sleeping*. Once that thread returns (either after successfully acquiring the desired resource or after a timeout), SCALENE resets the status of the calling thread to *executing*. SCALENE only attributes time to currently executing threads.

Now, when the main thread receives a signal, SCALENE invokes `threading.enumerate()` to collect a list of all running threads. It then obtains the Python stack frame from each thread using Python’s `sys._current_frames()` method. As above, SCALENE walks the stack to find the appropriate line of code to attribute execution time.

Finally, SCALENE uses bytecode disassembly (via the `dis` module) to distinguish between time spent in Python vs. native code. Whenever Python invokes an external function, it does so using a bytecode whose textual representation is either `CALL_FUNCTION`, `CALL_METHOD`, or, as of Python 3.11, `CALL`. SCALENE builds a map of all such bytecodes at startup.

SCALENE checks the stack of each running thread to see if the currently executing bytecode is a call instruction. SCALENE uses this information to infer if the thread is currently executing Python or native code.

If a thread is running Python code, it is likely to spend almost no time in a bytecode before executing another Python bytecode. By contrast, if it is running native code, it will be “stuck” on the `CALL` bytecode for the duration of native execution. Leveraging this fact lets SCALENE accurately attribute execution time: if it finds that a stack is executing `CALL`, SCALENE assigns time elapsed to the native counter; otherwise, it assigns time elapsed to the Python counter.



**Figure 4: Threshold-Based vs. Rate-Based Sampling.** SCALENE employs a novel sampling scheme that only triggers when memory use grows or declines beyond a set threshold, letting it capture all significant changes in footprint (beyond a given granularity, here 10MB) with low overhead. (§3.2).

## 3 Memory and Copy Volume Profiling

Almost all past profilers either report CPU time or memory consumption; SCALENE reports both, at a line granularity. It is vital that SCALENE track memory both inside Python and out, as external libraries are often responsible for a considerable fraction of memory consumption.

### 3.1 Intercepting Allocation Calls

SCALENE intercepts all system allocator calls (`malloc`, `free`, etc.) as well as Python internal memory allocator by inserting its own “shim” memory allocator, using Python’s built-in memory hooks. This two-fold approach lets SCALENE distinguish between native memory allocated by libraries and Python memory allocated in the interpreter.

The shim allocator extends and uses code from the Heap Layers memory allocator infrastructure [5]; SCALENE injects it via library interposition before Python begins executing using `LD_PRELOAD` on Linux and `DYLD_INSERT_LIBRARIES` on Mac OS X. To interpose on Python’s internal memory allocator, SCALENE uses Python’s custom allocator API (`PyMem_SetAllocator`).

Each shim allocator function handles calls by sampling for inclusion in the profiling statistics (§3.2) and then passing these to the original (Python or system) allocator. A complication arises from the fact that the Python allocators themselves may handle allocation requests by calling into the system allocator. To avoid counting Python allocations also as native allocations, SCALENE sets a flag, stored in thread-specific data, indicating it is within a memory allocator. When a shim allocator function is called with this flag set, it skips over the profiling, just forwarding to the original allocator. This approach both avoids double counting and simplifies writing profiling code, as it can allocate memory normally without causing infinite recursion.

## 3.2 Threshold-Based Sampling

The standard approach to sampling memory profilers, as exemplified by several non-Python memory profilers in Android, Chrome, Go, and Google's `tcmalloc` [40] and in Java TLAB based sampling [1], use a *rate-based* sampling approach. This sampler triggers samples at a rate proportional to the number of bytes allocated or freed. In effect, each byte allocated or freed corresponds to a Bernoulli trial with a given probability  $p$  of sampling; e.g., if  $p = 1/T$ , then (in expectation) there will be one sample per  $T$  bytes. In practice, for efficiency, these samplers initialize counters to random numbers drawn from a Poisson process or a geometric distribution with the same parameter. Each allocation and free then decrements this counter by the number of bytes allocated and freed, and triggers a sample when the counter drops below 0.

By contrast, the SCALENE sampler introduces *threshold-based* sampling. The allocator maintains a count of all memory allocations and frees, in bytes. Once the absolute difference between allocations and frees crosses a threshold ( $|A - F| \geq T$ ), SCALENE triggers a sample, corresponding to appending an entry to a sampling file and resets the counters. Figure 4 illustrates this operation. The sampling threshold  $T$  is currently set to a prime number slightly above 10MB; SCALENE uses a prime number to reduce the risk of stride behavior interfering with sampling.

Threshold-based sampling has several advantages over rate-based sampling. Unlike rate-based sampling, which is triggered by all allocation activity (even when it has almost no effect on footprint), threshold-based sampling is only triggered by significant memory use growth or decline. Table 2 shows the dramatic reduction in the number of samples, as high as  $676\times$  (median:  $18\times$ ) fewer. This reduced number of samples translates directly to lower runtime overhead.

At the same time, threshold-based sampling deterministically triggers a new sample whenever a significant change in footprint occurs. This approach improves repeatability over rate-based sampling (which is probabilistic) and avoids the risk of missing these changes.

Crucially, threshold-based sampling avoids two sources of bias inherent to rate-based sampling. Rate-based sampling can overstate the importance of allocations that do not contribute to an increased footprint since it does not take memory reclamation or footprint into account. It also biases the attribution of memory consumption to lines of code running Python code that exercises the allocator, rather than code responsible for footprint changes. By contrast, threshold-based sampling filters out the vast number of short-lived objects that are created by the Python interpreter itself, and only triggers based on events that change footprint.

## 3.3 Collecting and Processing Samples

When a memory sample is taken, SCALENE temporarily enables tracing using Python's `PyEval_SetTrace`. Tracing remains active only until it detects execution has moved on from that line. This approach lets SCALENE properly account for average memory consumption per line.

Each entry in SCALENE's sampling file includes information about allocations or frees, the fraction of Python (vs. native) allocations in the total sample, as well as an attribution to a line of Python source code.

SCALENE attributes each sample to Python source code at the time the sample is taken. It does so by obtaining the current thread's call stack from the interpreter and skipping over frames until one within profiled source code is found. This attribution needs to happen whenever a sample is taken, so it is implemented as a C++ extension module, using read-only accesses to Python structures. SCALENE loads this module upon startup, which in turn uses a symbol exported by the shim library to complete the linkage, making itself available to the shim.

A background thread in SCALENE's Python code reads from the sampling file and updates the profiling statistics. SCALENE also tracks the current memory footprint, which it uses both to report maximum memory consumption and memory trends. SCALENE records a timestamp and the current footprint at each threshold crossing, which SCALENE uses to generate memory trend visualizations (§5).

## 3.4 Memory Leak Detection

Like other garbage-collected languages, Python can suffer from memory leaks when references to objects are accidentally retained so that the garbage collector cannot reclaim them. As in other garbage collected languages, identifying leaks in Python programs is generally a slow, manual process.

In Python, the standard approach to identifying leaks is to first activate `tracemalloc`, which records the size, allocation site, and stack frame for each allocated object. The programmer then inserts calls at the appropriate place to produce a series of heap snapshots, and then manually inspects snapshot diffs to identify growing objects. This approach is laborious and depends on a post hoc analysis of the heap. It also can be quite slow. In our tests, just activating `tracemalloc` can slow Python applications down by  $4\times$ .

SCALENE incorporates a novel sampling-based memory leak detection algorithm that is both simple and efficient. The algorithm piggybacks on threshold-based sampling (§3.2). Whenever the sampler triggers because of memory growth, SCALENE checks if this growth has reached a new maximum footprint. If so, SCALENE records the sampled allocation. Every call to free then checks to see whether this object is ever reclaimed. This checking is cheap—a single pointer comparison—and highly predictable (almost always false).

**Leak Score:** At the next crossing of a maximum, SCALENE updates a *leak score* for the sampled object. The leak score tracks the historic likelihood of reclamation of the sampled object, and consists of a pair of (*freed*, *mallocs*). SCALENE first increments the *mallocs* field when it starts tracking an object, and then increments the *freed* field only if it reclaimed the allocated object. It then resumes tracking with a newly sampled object.

Intuitively, leak scores capture the likelihood that an allocation site is leaking. A site with a high number of *mallocs* and no *freed* is a plausible leak. By contrast, a site with a matching number of *mallocs* and *freed* is probably not a leak. The more observations we make, the higher the likelihood that we are observing or ruling out a leak.

SCALENE uses Laplace’s Rule of Succession to compute the likelihood of a success or failure in the next Bernoulli trial, given a history of successes and failures [50]. Here, successes correspond to reclamations (*freed*) and failures are non-reclamations (*mallocs* - *freed*). According to the Rule of Succession, SCALENE computes the leak probability as  $1.0 - (\text{freed} + 1) / (\text{mallocs} - \text{freed} + 2)$ .

**Leak Report Filtering and Prioritization:** To provide maximal assistance to Python developers, SCALENE filters and augments leak information. First, to limit the number of leak reports, SCALENE only reports leaks whose likelihood exceeds a 95% threshold, and only when the slope of overall memory growth is at least 1%. Second, SCALENE lets developers prioritize leaks by associating each leak with an estimated *leak rate*: the average amount of memory allocated at a given line divided by time elapsed.

### 3.5 Copy Volume

SCALENE uses sampling to collect information about *copy volume* (megabytes per second of copying) by line. This metric, which SCALENE introduces, helps identify costly (and often inadvertent) copying across the Python/native divide, or copying between CPU and GPU.

The SCALENE shim library used for memory allocation also interposes on *memcpy*, which is invoked both for general copying (including to and from the GPU, and copying across the Python/native boundary). As with memory allocations, SCALENE writes an entry to a sampling file once a threshold number of bytes has been copied. However, unlike memory sampling, copy volume sampling employs classical rate-based sampling: since copy volume only ever increases, threshold-based sampling and rate-based sampling would effectively be equivalent. The current *memcpy* sampling rate is set at a multiple of the allocation sampling rate.

## 4 GPU Profiling

SCALENE performs both line-granularity GPU utilization and memory profiling on systems equipped with NVIDIA GPUs. This feature helps Python programmers identify whether they are efficiently making use of their GPUs.

SCALENE piggybacks GPU sampling on top of its CPU sampler. Every time SCALENE obtains a CPU sample, it also collects the total currently used GPU memory and utilization, which it associates with the currently executing line of code. Whenever possible, SCALENE employs per-process ID accounting, which increases accuracy in a shared GPU setting.

At startup, SCALENE checks to see if per-process ID accounting has been enabled on the attached NVIDIA GPU. If not, SCALENE offers to enable it, a process that requires that the user invoke SCALENE once with super-user privileges.

## 5 GUI Design and Implementation

SCALENE’s primary user interface is web-based, though it also offers a non-interactive rich text-based CLI. In the UI, SCALENE not only reports net memory consumption per line, but also reports memory usage over time, both for the program as a whole and for each individual line. Figure 2 presents several examples. The x-axis corresponds to execution time, and the y-axis corresponds to the footprint of the program, as seen by that line of code.

Because it can be expensive to visualize graphs with large numbers of points, SCALENE limits the number of points it outputs in its JSON payload and HTML output. Prior to generating the profile output, SCALENE applies the Ramer-Douglas-Peucker (RDP) algorithm [9, 32] to each line’s memory footprint log (if any). The RDP algorithm aims to reduce the total number of points while preserving the overall shape of the curve. The RDP algorithm depends on a parameter  $\epsilon$ , a distance parameter below which RDP merges adjacent points; SCALENE sets  $\epsilon$  to a value that approximately reduces the total number of points to a manageable size (100 points). Sometimes this process fails to reduce the number of points sufficiently. To guarantee that the number of points is always bounded, after applying RDP, SCALENE randomly downsamples all memory logs to exactly 100 points.

To further ensure the scalability of the user interface, SCALENE only reports lines of code that are responsible for at least 1% of execution time (CPU or GPU) or at least 1% of total memory consumption, along with the preceding and following line. This approach guarantees that a SCALENE profile never contains more than 300 lines. In practice, profiles are generally skewed and resulting profilers are often far smaller.

## 6 Evaluation

Our evaluation answers the following questions: How does SCALENE’s CPU profiling accuracy compare to other CPU profilers? (§6.2) How does SCALENE’s memory profiling accuracy compare to other memory profilers? (§6.3) How does SCALENE’s CPU profiling overhead compare to other CPU profilers? (§6.4) How does SCALENE’s memory profiling overhead compare to other memory profilers? (§6.5)

### 6.1 Experimental Setup

Our prototype of SCALENE consists of roughly 3,500 lines of Python 3 code and 1,700 lines of C++-17 code; its user interface comprises 800 lines of JavaScript, excluding white space and comments as measured by `cloc` [8]. This prototype runs on Linux, Microsoft Windows, and Mac OS X, for Python versions 3.8 and higher; we report Linux results here. We use the latest version of SCALENE, released 12/08/2022.

We perform all experiments on an 8-core 4674 MHz AMD Ryzen 7, equipped with 32GB of RAM and an NVIDIA GeForce RTX 2070 GPU, running Linux 5.13.0-35-generic. All C/C++ code is compiled with `g++` version 9, and we use CPython version 3.10.9 (release date 12/06/2022) For overhead numbers, we report the interquartile mean of 10 runs.

### 6.2 CPU Profiling Accuracy

Here, we explore a specific threat to the accuracy of Python CPU profilers. We show that some profilers suffer from a *probe effect* that distorts the time spent by applications. Specifically, we observe that Python profilers that rely on Python’s tracing facility exhibit a bias caused by tracing triggering both on function calls and lines of code, dilating the apparent time spent in function calls. We call this phenomenon *function bias*; we show that sampling-based profilers like SCALENE do not suffer from this bias.

We wrote a microbenchmark to measure this bias. It executes a varying number of iterations of two semantically identical functions: one invokes another function inside its loop, while the other inlines the same logic. We vary the amount of time spent in one function versus the other, and compare the profiler results to the ground truth, as measured with high resolution timers.

Figure 5 presents the results of this experiment. The x-axis is the amount of time measured while running the variant with a function call (the ground truth), while the y-axis is the amount of time reported by each profiler. The ideal is a diagonal running from the origin. The trace-based profilers exhibit a high degree of inaccuracy, showing significant function bias. In the worst case, one such profiler reports a function takes 80% of execution time while in fact it only consumes 25%. We conclude that such profilers may be too potentially misleading to be of practical value for developers.

Accuracy: Time spent vs. time reported

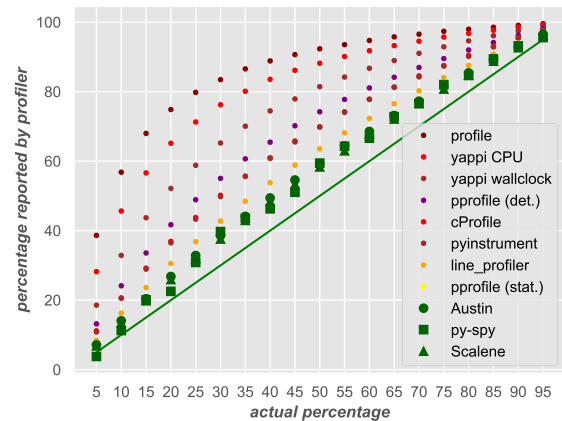


Figure 5: CPU Profiling Accuracy: SCALENE is among the most accurate CPU profilers. This graph measures the accuracy of profile reports vs. the actual time spent in functions; the ideal is shown by the diagonal line (the amount the profiler reports is exactly the time spent). Some profilers are highly inaccurate (§6.2).

### Memory accounting, Scalene vs. RSS-based proxies

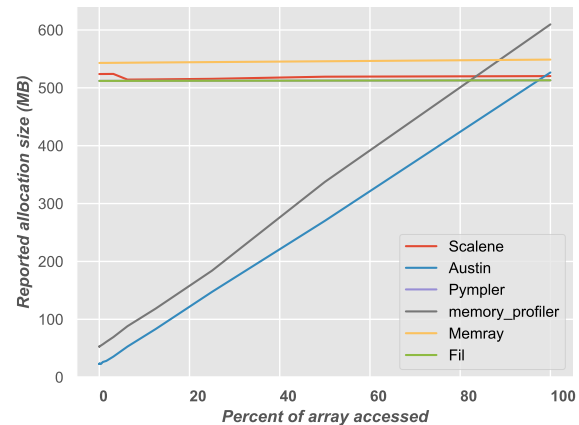


Figure 6: Memory Profiling Accuracy: SCALENE produces more accurate memory profiles than resident set size (RSS) based profilers. Varying the amount of memory accessed causes RSS-based profilers to significantly under-report, and sometimes over-report, the true amount of allocated memory. Interposition-based profilers are far more accurate (§6.3).

### 6.3 Memory Profiling Accuracy

We next compare the accuracy of memory profilers with a simple test designed to explore the effect of using resident set size (RSS) instead of direct memory tracking. We expect to see a difference between the two, since RSS corresponds to the use of memory rather than the allocation of objects. Our test first allocates a single 512MB array, and then accesses a varying amount of the array (from 0% to 100%).

Figure 6 presents the result, confirming our hypothesis. Both `memory_profiler` and `Austin` rely on RSS as a proxy



Benchmark	Repetitions	Time
async_tree_io <sub>none</sub>	22	11.9s
async_tree_io <sub>io</sub>	9	12.0s
async_tree_io <sub>cpu_io_mixed</sub>	14	12.3s
async_tree_io <sub>memoization</sub>	16	10.6s
docutils	5	12.5s
fannkuch	3	12.1s
mdp	5	13.4s
pprint	7	12.8s
raytrace	25	11.1s
sympy	25	11.3s

**Table 1: Benchmark suite:** We conduct our evaluation using the top ten most time consuming benchmarks from the standard pyperformance benchmark suite. For each, we extend their running time by running them in a loop enough times to exceed 10 seconds.

for memory consumption. But RSS is a measure of the *physical* memory currently in use, and depends on a number of other factors such as memory access patterns and the memory needs of other processes. The figure clearly shows that this proxy can be wildly inaccurate, leading to under-reporting and even over-reporting the size of the allocated object. The other profilers directly measure allocation, and produce much more accurate results. Both SCALENE and Fil report within 1% of the actual size of the allocated object (512MB), while Memray is within 6%.

**Drawbacks of peak-only profiling:** Both Fil and Memray only report live objects at the point of peak memory allocation by a program. This information can be useful, but it can both exaggerate the potential for reducing memory and obscure other sources of memory consumption. Consider a program that allocates and discards a 4GB object, and then allocates a 4GB + 8 byte object. A report that only contains information at the point of peak allocation will reveal the second object but not the first. That profile will suggest an enormous opportunity to save memory, but eliminating the second object entirely would have almost no effect on peak memory consumption. Unlike peak profilers, SCALENE provides information about *all* significant memory allocation over time, giving programmers a global view of memory consumption.

**Summary:** SCALENE’s memory profiling is highly accurate, while capturing memory consumption over time.

## 6.4 CPU Profiling Overhead

In our evaluation, we use the ten longest-running benchmarks from pyperformance, the standard suite for evaluating Python performance (Figure 1). We modify these benchmarks to run in a loop so that they execute for at least 10 seconds on our experimental platform. We also modify the benchmarks slightly by adding @profile decorators, as these are

Benchmark	Rate	Threshold	Ratio
async_tree_io <sub>none</sub>	556	215	3×
async_tree_io <sub>io</sub>	524	187	3×
async_tree_io <sub>cpu_io_mixed</sub>	719	167	4×
async_tree_io <sub>memoization</sub>	375	167	2×
docutils	20	5	4×
fannkuch	426	5	85×
mdp	316	6	53×
pprint	7976	23	347×
raytrace	215	7	31×
sympy	6757	10	676×
<b>Median:</b>			18×

**Table 2: Threshold vs. Rate-Based Sampling:** SCALENE’s threshold-based sampling tracks footprint with as many as 676× fewer samples than conventional rate-based sampling (median: 18×).

required by some profilers; we also add code to ignore the decorators when they are not used. Finally, we add a call to `system.exit(-1)` to force py-spy to generate output. Figure 7 provides the results of running the profilers across all these benchmarks.

**Summary:** In general, SCALENE imposes low to modest overhead (median: 2% for CPU+GPU, and 30% for full functionality), placing it among the profilers with the lowest overhead.

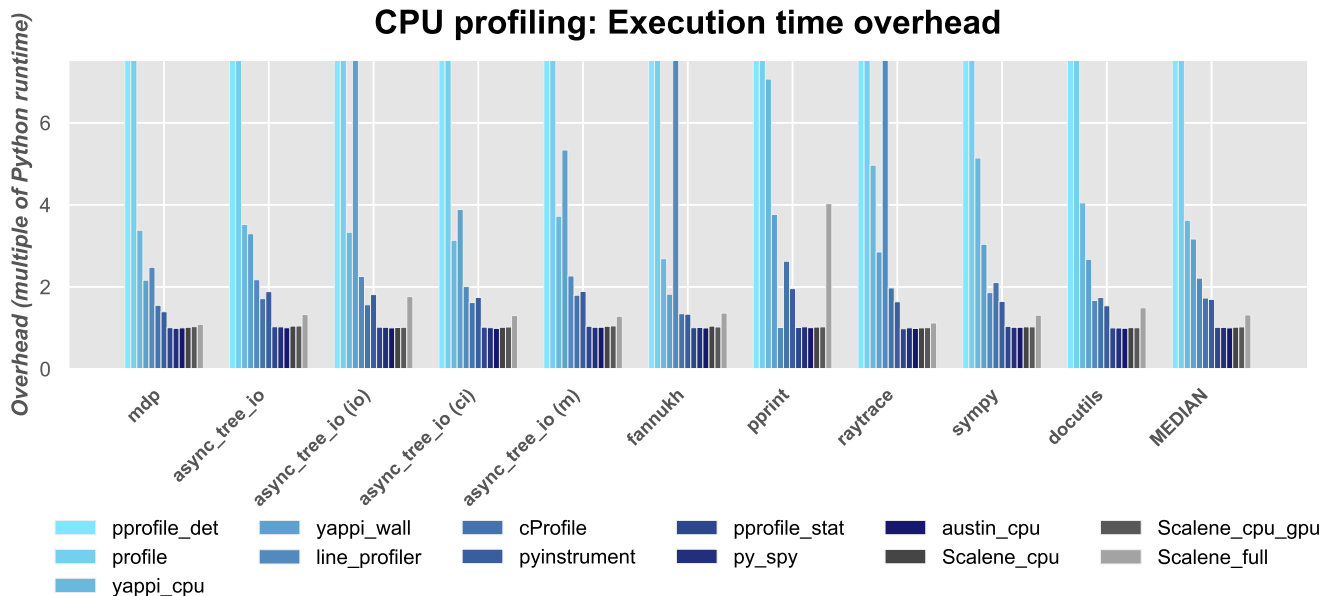
## 6.5 Memory Profiling Overhead

Next, we evaluate the overhead of memory profilers (`memory_profiler`, Fil, Memray and Austin), and compare them to SCALENE. We use the same benchmarks as we used for measuring runtime overhead for CPU profilers.

Figure 8 shows the results. Because it can slow down execution by at least 150×, we omit `memory_profiler` from the graph. SCALENE’s performance is competitive with the other profilers; while Austin is faster, as Section 6.3 shows, it provides inaccurate estimates of memory consumption.

**Log file growth:** Some memory profilers feature a surprising other source of overhead. Two of the memory profilers, Memray and Austin, produce detailed (and copious) logs of memory activity that may limit their usefulness for profiling long-lived applications.

Memray deterministically logs information including all allocations, all updates to the Python stack, and context switches, which it later post-processes for reporting. Austin similarly generates logs meant to be consumed by an external tool. These files can grow rapidly: in our tests, Memray’s output file grows by roughly 3MB/second, while Austin’s grows by 2MB/second.



**Figure 7: CPU profiling: SCALENE has modest overhead.** Despite collecting far more detailed information, SCALENE is competitive with the best-of-breed CPU profilers in terms of overhead (§6.4). The graph truncates the slowest profilers; see Table 3 for full data.

	mdp	a_t_i	(io)	(ci)	(m)	fannkuch	pprint	raytrace	sympy	docutils	Median
py_spy	0.99×	1.03×	1.02×	1.01×	1.02×	1.01×	1.03×	1.00×	1.02×	1.00×	1.02×
cProfile	1.55×	1.71×	1.57×	1.62×	1.80×	1.35×	2.63×	1.98×	2.11×	1.74×	1.73×
yappi_wall	2.16×	3.30×	33.25×	3.89×	5.34×	1.82×	3.77×	2.85×	3.04×	2.67×	3.17×
yappi_cpu	3.38×	3.52×	3.33×	3.14×	3.72×	2.69×	7.07×	4.97×	5.14×	4.05×	3.62×
pprofile_stat	1.01×	1.03×	1.02×	1.02×	1.04×	1.01×	1.01×	0.98×	1.04×	1.00×	1.02×
pprofile_det	37.80×	35.06×	29.30×	28.09×	35.85×	65.19×	103.73×	56.23×	55.68×	34.78×	36.83×
line_profiler	2.48×	2.18×	2.25×	2.01×	2.27×	8.92×	1.01×	11.59×	1.86×	1.67×	2.21×
profile	14.30×	14.53×	13.54×	12.48×	15.71×	10.41×	55.68×	20.87×	26.17×	15.66×	15.1×
pyinstrument	1.40×	1.89×	1.81×	1.74×	1.89×	1.34×	1.96×	1.64×	1.65×	1.54×	1.69×
austin_cpu	1.00×	1.01×	1.00×	0.99×	1.02×	1.00×	1.01×	0.99×	1.02×	0.99×	1.00×
austin_full	0.98×	1.01×	0.99×	1.00×	1.01×	1.01×	1.01×	1.00×	1.00×	0.99×	1.00×
memray	2.43×	4.34×	3.21×	4.80×	3.85×	2.92×	5.36×	3.21×	4.12×	4.11×	3.98×
fil	1.75×	3.05×	2.76×	2.73×	2.91×	1.85×	2.88×	2.15×	2.58×	2.68×	2.71×
memory_profiler	> 150×	37.90×	28.42×	36.32×	41.90×	> 150×	1.01×	> 150×	18.95×	9.19×	37.11×
Scalene_cpu	1.02×	1.05×	1.01×	1.02×	1.04×	1.05×	1.02×	1.00×	1.03×	1.01×	1.02×
Scalene_cpu_gpu	1.03×	1.05×	1.02×	1.02×	1.05×	1.02×	1.03×	1.01×	1.03×	1.01×	1.02×
Scalene_full	1.09×	1.33×	1.76×	1.30×	1.28×	1.36×	4.03×	1.12×	1.31×	1.49×	1.32×

**Table 3: Detailed profiling overhead (CPU and memory).** All numbers are relative to the Python baseline (no profiling); a\_t\_i refers to the async\_tree\_io benchmark.

By contrast, SCALENE only records samples when memory consumption grows or shrinks by a large amount (§3.2), leading to vastly smaller logs. For example, when running the mdp benchmark, Austin’s log file consumes 27MB and Memray’s log file consumes almost 100MB, while SCALENE’s log file consumes just 32K.

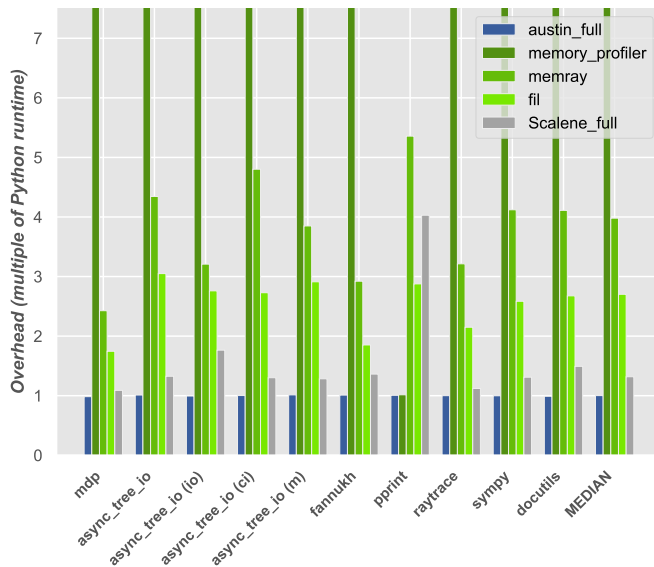
**Summary:** Among the accurate memory profilers, SCALENE operates with the lowest overhead (median: 1.32× vs. 3.98× (memray) and 2.71× (Fil)), while capturing memory usage over time and producing small log files.

## 7 Case Studies

This section includes reports on real-world experience by external developers using SCALENE to identify and resolve performance issues. For each, we identify the features of SCALENE that were instrumental in enabling these optimizations.

**Rich:** A user reported severe slowness when printing large tables to the developer of Rich [20], an immensely popular Python library for formatting text in the terminal (downloaded over 130 million times, with 41K stars on GitHub). When Rich’s developer profiled it using SCALENE, he identified two lines occupying a disproportional amount of run-

### Memory profiling: Execution time overhead



**Figure 8: Memory profiling overhead: SCALENE has competitive runtime overhead.** Despite collecting far more detailed information, SCALENE is faster than the accurate memory profilers (§6.5).

time. SCALENE indicated that a call to `isinstance` was taking an unexpectedly large amount of time—though each call takes very little time, the developer reported that it was being called 80,000 times. Rich’s developer replaced these calls with a lower-cost function, `hasattr`. In our benchmarks, `isinstance` (when marked as a runtime protocol via `@typing.runtime_checkable`) can run over  $20\times$  slower than `hasattr`. The developer also indicated that an unnecessary copy was being performed once every cell. Optimizing these calls led to a reported 45% improvement in runtime when rendering a large table. **[Features: Fine-grained CPU profiling, copy volume.]**

**Pandas – Chained Indexing:** A developer was seeing suboptimal performance in their code using Pandas [41]. SCALENE identified that a list comprehension performing nested indexes into a Pandas dataframe was taking an unexpectedly large amount of time and resulting in a significant amount of copy volume. The developer noted that the first level of indexing was repeatedly using a string that was loop invariant; the way this was being done in Pandas caused it to perform copies rather than using views, a problem known as chained indexing ([https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing.html#returning-a-view-versus-a-copy](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)). After manually hoisting this outer indexing operation, the developer obtained an  $18\times$  speedup. **[Features: Copy volume and fine-grained CPU profiling.]**

**Pandas – concat and groupby queries:** An instructor had their students use SCALENE in a tutorial designed to teach higher performance Pandas. The instructor found that SCALENE revealed significant issues in both performance and space consumption when using Pandas. First, SCALENE revealed that calling `concat` on Pandas dataframes was using more memory than anticipated. SCALENE’s copy volume reporting revealed that the problem was that `concat` copies all the data by default (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html#pandas.concat>), effectively doubling memory usage when managing large dataframes. Second, SCALENE confirmed that excessive RAM usage in some groupby operations is due to copying of the groups; this bug has been reported to the Pandas developers (<https://github.com/pandas-dev/pandas/issues/37139>). Restructuring the groupby operation reduced memory consumption by a further 1.6GB (43%). **[Features: Fine-grained CPU and memory profiling, copy volume.]**

**NumPy vectorization:** A graduate student was using NumPy to implement classification with gradient descent and was seeing extremely low performance. SCALENE showed that 99% of the time was being spent in Python (rather than native code), indicating that his code was not vectorized. In other words, the code was not expressed in a way that allowed NumPy to efficiently compute vector operations (using native code). Guided by SCALENE’s feedback, the graduate student gradually improved the performance from 80 iterations per minute to 10,000 per minute, a  $125\times$  improvement. **[Feature: Fine-grained native vs. Python CPU profiling.]**

**Semantic Scholar:** Semantic Scholar reports that they have been using SCALENE as part of their tool suite for operationalizing their machine learning models. Recently, they found that a model was cost-prohibitive and put an entire product direction in jeopardy. They generated a set of test data and ran their models with SCALENE. SCALENE’s output was able to pinpoint the issues and help them validate that their changes were having an impact. While iteratively using SCALENE while applying optimizations, they were ultimately able to reduce costs by 92%. Additionally, SCALENE allowed Semantic Scholar’s developers to quickly determine what fraction of their runtime would benefit from hardware acceleration and what CPU-bound code they needed to optimize in order to achieve their goals. **[Features: Simultaneous, fine-grained CPU, memory, and GPU profiling.]**

**Summary:** In nearly all of the cases described above, SCALENE was either invaluable or provided additional help that narrowed down performance issues, by leveraging unique or novel features of SCALENE: separation of native from Python

time, copy volume, GPU profiling, and its ability to simultaneously measure memory and CPU usage. Though other tools can separately identify high RAM usage or slow code, past tools would either misattribute the location of usage due to the use of resident set size as a metric (unlike SCALENE’s accurate memory profiling approach) or not be able to simultaneously measure memory usage and CPU usage. The insights generated by SCALENE were actionable, yielding substantial improvements in execution time and space, and reducing cost.

## 8 Related Work

There is an extensive history of profilers; we focus our attention here on profilers that specifically support Python. The Python ecosystem contains many Python profilers, most of which have not been discussed in the academic literature. This section describes the most prominent profilers; Figure 1 provides a tabular overview.

We first survey CPU-only profilers. We divide them into two categories: *deterministic* (tracing-based) (§8.1) and *sampling-based* (§8.2). We then discuss memory profilers (§8.3), ML-specific profilers (§8.4), other Python profilers (§8.5), and general profilers with Python support (§8.6), and touch on more distantly related profilers for other languages (§8.7).

### 8.1 Deterministic CPU profilers

Python provides built-in tracing support (`sys.settrace`) that several profilers build upon. The tracing facility, when activated, triggers a callback in response to a variety of events, including function calls and execution of each line of code. This deterministic, instrumentation-based approach leads to significant inaccuracies due to its probe effect, as Section 6.2 shows. Because of the overhead of tracing, they are also the slowest profilers.

**Function-granularity:** Python includes two built-in function-granularity profilers, `profile` [35] and `cProfile` [34]. The primary difference between these two profilers is that `cProfile`’s callback function is implemented in C, making it much faster (1.7× slowdown vs. 15.1×) and somewhat more accurate than `profile`. Another profiler, `yappi`, operates in two modes, wall clock time (sample-based) and CPU time (deterministic); it is among the most inaccurate of CPU profilers, with slowdowns ranging from 1.8× to 33.3×.

**Line-granularity:** `pprofile` [30] comes in two flavors: a deterministic and a “statistical” (sampling-based) profiler. Both flavors correctly work for multithreaded Python programs, unlike `line_profiler` [17]. All of these report infor-

mation at a line granularity. `pprofile_det` imposes a median overhead of 36.8×, while `line_profiler`’s median overhead is 2.2×.

### 8.2 Sampling-based CPU profilers

Sampling-based profilers are more efficient and often more accurate than the deterministic profilers. These include `pprofile_stat`, `py-spy` [10], and `pyinstrument` [33]. Their overhead is between 1× and 1.7×, comparable to SCALENE. Because it fails to cope with Python’s deferred signal delivery, `pprofile_stat` incorrectly ascribes zero runtime to execution of native code or code in child threads (§2).

Compared to past CPU-only profilers, SCALENE is nearly as fast or faster, more accurate, and provides more detailed CPU-related information, breaking down time spent into Python, native, and system time.

### 8.3 Memory profilers

`memory_profiler` is a deterministic memory profiler that uses Python’s trace facility to trigger it after every line of execution [36]. By default, it measures the RSS after each line executes and records the change from the previous line. `memory_profiler` also does not support Python applications using threads or multiprocessing.

`Fil` measures the peak allocation of the profiled program by interposing on system allocator functions and forcing Python to use the system allocator (instead of Python’s `Pymalloc`) [45]. `Fil` records a full stack trace whenever the current memory footprint exceeds a previous maximum. On exit, it produces a flamegraph [13] of call stacks responsible for memory allocation at the point of maximum memory consumption. The `Fil` website reports that it supports threads (“In general, `Fil` will track allocations in threads correctly.” [46]). However, in our tests, `Fil` (version 2022.6.0) fails to ascribe any memory allocations to threads. `Fil` also does not currently support multiprocessing.

`Memray` is a recently released (April 2022), Linux-only memory profiler that deterministically tracks allocations and other profiler events [37]. `Memray` interposes upon the C allocation functions and optionally on the `pymem` functions, letting it distinguish native from Python allocations.

The only previous CPU+memory profiler we are aware of besides SCALENE is `Austin` [43]. `Austin` reduces performance overhead by profiling with a separate process.

### 8.4 Profilers for Machine Learning Libraries

Two widely used machine learning libraries, TensorFlow and PyTorch [28], include their own profilers [11, 19]. Both profilers are targeted at identifying performance issues specific to deep learning training and inference. For example, the



PyTorch profiler can attribute runtime to individual operators (running inside PyTorch’s native code). NVIDIA’s Deep Learning profiler (DLprof) [23] provides similar functionality for either PyTorch or TensorFlow. Unlike SCALENE, these profilers are specific to machine learning workloads and are not suitable for profiling arbitrary Python code. These profilers are complementary to SCALENE, which aims to be a general-purpose profiler. They also lack many of SCALENE’s features.

## 8.5 Other Python Profilers

PieProf aims to identify and surface specific types of inefficient interactions between Python and native code [39]. PieProf leverages data gathered from on-chip performance monitoring units and debug registers combined with data from libunwind and the Python interpreter to identify redundant loads and stores initiated by user-controlled code. It surfaces pairs of redundant loads and stores for the developer to potentially optimize. PieProf is not publicly available, so it was not possible to empirically compare it to SCALENE.

## 8.6 Profilers with Python Support

Several non-Python specific conventional profilers offer limited support for Python. Intel’s VTune profiler [49] can attribute its metrics to Python lines, with a number of caveats, including “if your application has very low stack depth, which includes called functions and imported modules, the VTune Profiler does not collect Python data.” [16]. VTune does not directly distinguish between time spent in Python code and time spent in native code and does not track Python memory allocations. Google Cloud Profiler [12] only profiles Python execution time, but neither distinguishes between Python and native time nor does it perform memory profiling for Python. Both lack most of SCALENE’s other features.

Python 3.12, the current development version of Python, recently (November 2022) added support for use with the perf profiler on Linux platforms by reporting function names in traces [26]. Using perf in this mode only measures performance counters or execution time. Unlike SCALENE, perf does not measure memory allocation, or attribute runtime (Python or native) to individual lines of Python code.

## 8.7 Non-Python Profilers

AsyncProfiler is a Java profiler that, like SCALENE, profiles both CPU and memory [27]. AsyncProfiler is a sampling profiler that avoids the *safepoint bias problem* [21]. Since Python does not have safepoints (all garbage collection happens while the global interpreter lock is held), Python profilers cannot suffer from this bias. Instead, as we show, they can suffer from function bias (§6.2). Similarly, pprof is a profiler

for the Go language that can report both CPU and memory [7]. Both profilers use rate-based memory sampling (§3.2).

## 9 Conclusion

This paper presents SCALENE, a novel Python profiler. SCALENE delivers more actionable information than past profilers, all with high accuracy and low overhead. Its suite of novel algorithms enables SCALENE’s holistic reporting of Python execution. SCALENE has been released as open source at <https://github.com/plasma-umass/scalene>.

## Acknowledgements

We thank SCALENE’s users for their feature requests, questions, and bug reports, which have helped shape and guide this research. We are most grateful to users who contributed pull requests or worked with us to resolve compatibility issues, including Raphael Cohen, James Garity, Ryan Grout, Friday James, and Marguerite Leang. We thank the users who contributed their experiences, reported here as case studies: Will McGugan, Ian Oszvald, Donald Pinckney, Nicolas van Kempen, and Chris Wilhelm. Finally, we thank our shepherd, Phil Levis, and the reviewers of this paper, whose feedback helped improve not only the paper but also SCALENE itself.

This material is based upon work supported by the National Science Foundation under Grant No. 1954830.

## References

- [1] Improved JFR allocation profiling in JDK 16. <https://withent.blogspot.com/2021/01/improved-jfr-allocation-profiling-in.html>, Jan. 2021.
- [2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

- [4] M. Belanger and D. Deville. How we rolled out one of the largest Python 3 migrations ever - Dropbox. <https://dropbox.tech/application/how-we-rolled-out-one-of-the-largest-python-3-migrations-ever>, Sept. 2018.
- [5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In M. Burke and M. L. Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 114–124. ACM, 2001.
- [6] S. Cass. Top programming languages 2022. <https://spectrum.ieee.org/top-programming-languages-2022>, Aug. 2022.
- [7] R. Cox and S. Ma. Profiling go programs. <https://go.dev/blog/pprof>, May 2013.
- [8] A. Danial. cloc: v1.94. <https://github.com/AlDanial/cloc>, July 2022.
- [9] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.
- [10] B. Frederickson. py-spy: Sampling profiler for Python programs. <https://github.com/benfred/py-spy>.
- [11] Google Corporation. Optimize TensorFlow performance using the Profiler. <https://www.tensorflow.org/guide/profiler>.
- [12] Google LLC. Google Cloud: Profiling Python applications. <https://cloud.google.com/profiler/docs/profiling-python>, 2022.
- [13] B. Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, June 2016.
- [14] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented programming, Systems, Languages, and Applications*, pages 313–326. New York, NY, USA, 2005. ACM Press.
- [15] Instagram Engineering. Python - Instagram Engineering. <https://instagram-engineering.com/tagged/python>, 2019.
- [16] Intel Corporation. Intel VTune Profiler User Guide. <https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/code-profiling-scenarios/python-code-analysis.html>, 2022.
- [17] R. Kern. line\_profiler: Line-by-line profiling for Python. [https://github.com/pyutils/line\\_profiler](https://github.com/pyutils/line_profiler).
- [18] R. Komorn. Python in production engineering. <https://engineering.fb.com/production-engineering/python-in-production-engineering/>, May 2016.
- [19] M. Lukyanov, G. Hua, G. Chauhan, and G. Dankel. Introducing PyTorch Profiler - the new and improved performance tool. <https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/>.
- [20] W. McGugan. Rich is a Python library for rich text and beautiful formatting in the terminal. <https://github.com/Textualize/rich>.
- [21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In B. G. Zorn and A. Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 187–197. ACM, 2010.
- [22] Netflix Technology Blog. Python at Netflix. <https://netflixtechblog.com/python-at-netflix-bba45dae649e>, Apr. 2019.
- [23] NVIDIA Corporation. NVIDIA Deep Learning Profiler. <https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/>.
- [24] S. O’Grady. The RedMonk Programming Language Rankings: June 2022. <https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22/>, Oct. 2022.
- [25] T. E. Oliphant. Guide to NumPy. <https://web.mit.edu/dvp/Public/numpybook.pdf>, 2006.
- [26] Pablo Galindo. Python support for the Linux perf profiler. [https://docs.python.org/3.12/howto/perf\\_profiling.html](https://docs.python.org/3.12/howto/perf_profiling.html), 2022.
- [27] A. Pangin. async-profiler. <https://github.com/jvm-profiling-tools/async-profiler>.
- [28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

- [29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, and D. Cournapeau. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [30] V. Pelletier. pprofile: Line-granularity, thread-aware deterministic and statistic pure-Python profiler. <https://github.com/vpelletier/pprofile>.
- [31] Python Software Foundation. Signals and threads. <https://docs.python.org/3/library/signal.html#signals-and-threads>.
- [32] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, 1(3):244–256, 1972.
- [33] J. Rickerby. pyinstrument: Call stack profiler for Python. <https://github.com/joerick/pyinstrument>.
- [34] B. Rosen and T. Czotter. The Python Profilers (cProfile). <https://docs.python.org/3.8/library/profile.html>.
- [35] J. Roskind. The Python Profilers (profile). <https://docs.python.org/3.8/library/profile.html>.
- [36] S. Saffron. memory\_profiler. [https://github.com/SamSaffron/memory\\_profiler](https://github.com/SamSaffron/memory_profiler).
- [37] P. G. Salgado. Memray. <https://bloomberg.github.io/memray/>.
- [38] Stack Overflow. Stack Overflow Developer Survey 2022. <https://survey.stackoverflow.co/2022/#most-popular-technologies-language>, May 2022.
- [39] J. Tan, Y. Chen, Z. Liu, B. Ren, S. L. Song, X. Shen, and X. Liu. Toward efficient interactions between Python and native libraries. In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1117–1128. ACM, 2021.
- [40] TCMalloc Team. How sampling in TCMalloc works. <https://github.com/google/tcmalloc/blob/master/docs/sampling.md>, 2022.
- [41] The Pandas development team. pandas-dev/pandas: Pandas. <https://github.com/pandas-dev/pandas>, July 2022. [Online; accessed 4-July-2022].
- [42] TIOBE Software BV. TIOBE Index for December 2022. <https://www.tiobe.com/tiobe-index/>, Dec. 2022.
- [43] G. N. Tornetta. austin: A frame stack sampler for cpython. <https://github.com/P403n1x87/austin>.
- [44] D. Trotter. Grumpy: Go running Python! <https://opensource.googleblog.com/2017/01/grumpy-go-running-python.html>, Jan. 2017.
- [45] I. Turner-Trauring. Fil profiler. <https://pythonspeed.com/fil/>.
- [46] I. Turner-Trauring. Threading in NumPy (BLAS), Zarr, numexpr. <https://pythonspeed.com/fil/docs/threading.html>.
- [47] G. van der Meer. How we use Python at Spotify. <https://labs.spotify.com/2013/03/20/how-we-use-python-at-spotify/>, Mar. 2013.
- [48] Wikipedia contributors. Cpython — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=CPython&oldid=1095361531>, 2022. [Online; accessed 4-July-2022].
- [49] Wikipedia contributors. VTune — Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=VTune&oldid=1107229725>, 2022. [Online; accessed 13-December-2022].
- [50] S. L. Zabell. The rule of succession. *Erkenntnis*, 31(2):283–321, 1989.



# Relational Debugging

## — Pinpointing Root Causes of Performance Problems

Xiang (Jenny) Ren<sup>1</sup>, Sitao Wang<sup>1</sup>, Zhuqi Jin<sup>1</sup>, David Lion<sup>1</sup>, Adrian Chiu<sup>1</sup>, Tianyin Xu<sup>2</sup>, and Ding Yuan<sup>1</sup>

<sup>1</sup>University of Toronto

<sup>2</sup>University of Illinois at Urbana-Champaign

### Abstract

Performance debugging is notoriously elusive—real-world performance problems are rarely clear-cut failures, but manifest through the accumulation of fine-grained symptoms. Oftentimes, it is challenging to determine performance anomalies—*absolute* measures are unreliable, as system performance is inherently *relative* to workloads. Existing techniques focus on identifying absolute predicates that deviate between executions, which limits their application to *performance* problems.

This paper introduces *relational debugging*, a new technique that automatically pinpoints the root causes of performance problems. The core idea is to capture and reason about relations between fine-grained runtime events. We show that relations provide immense utilities to explain performance anomalies and locate root causes. Relational debugging is highly effective with a minimal two executions (a good and a bad run), eliminating the pain point of producing and labeling many different executions required by traditional techniques.

We realize relational debugging by developing a practical tool named *Perspect*. *Perspect* directly operates on x86 binaries to accommodate real-world diagnosis scenarios. We evaluate *Perspect* on twelve challenging performance issues with various symptoms in Go runtime, MongoDB, Redis, and Coreutils. *Perspect* accurately located (or excluded) the root causes of these issues. In particular, we used *Perspect* to diagnose *two open bugs*, where developers failed to find root causes—the root causes reported by *Perspect* were confirmed by developers. A controlled user study shows that *Perspect* can speed up debugging by at least 10.87 times.

## 1 Introduction

Performance makes or breaks a software system: severe performance problems lead to unresponsiveness and even malfunctions; even seemingly-small performance degradations can incur high costs—a half-second search delay reduces Google’s revenue by 20% [33]. Therefore, it is crucial to diagnose performance problems in a timely manner.

Performance debugging is known to be elusive and difficult. Unlike functional failures with clear-cut symptoms, such as

crashes and runtime exceptions, performance problems are typically observed via the cumulative effect of fine-grained symptoms over time, such as latency increases due to regressions of code efficiency and resource overuse due to leaks. While fine-grained symptoms can potentially be identified by profilers [9, 10, 12, 13, 19], profiling alone cannot explain a performance anomaly—not every local symptom is related to the anomaly. Causality analysis [34, 37, 42] captures runtime events that are causally related to the symptoms, but it does not pinpoint the root causes in the code; the causality graph can be complex to navigate and analyze. In fact, it can be even challenging to determine whether or not the observed is performance an anomaly, because *absolute* measures are unreliable—system performance is inherently *relative* to inputs and workloads.

Existing performance diagnosis techniques target specific types of root causes and thus are limited when applied to many challenging performance problems. For example, X-ray [15] diagnoses performance anomalies due to unexpected inputs or configuration values by summarizing performance impact of each input/configuration value; however, as a tool designed for end users, X-ray does not address problems rooted in the source code. Statistical debugging [24, 26, 29, 32, 38] can address certain types of performance problems which result in differences in program predicates (e.g., branches and returns) [39]. However, unlike functional failures, many performance problems do not cause changes in predicates (e.g., due to distribution changes in runtime events). Besides, it can be challenging to design predicates and statistical models in the first place [39].

This paper introduces *relational debugging*, a new technique that automatically pinpoints the root causes of performance problems. The core idea is to capture *relations* between fine-grained runtime events. We show that relations provide immense utilities to explain performance anomalies and locate root causes. Relational debugging analogizes performance problems to *relative motion* in physics—just like the speed of an object is a relative measure depending on the reference frame, so is performance when viewed from different runtime events during program execution. Root causes of performance problems can be revealed by analyzing changes of



relative measures of these events (i.e., their relations) between a good run and a bad run (with performance anomalies).

Consider a real-world performance issue (see §2.1), where the developer observes an abnormal increase in memory consumption by a server application. Potential root causes can be: 1) an influx of more requests (in which consuming more memory is normal), 2) each request allocating more memory (indicating regression of code efficiency), and 3) allocated memory not being reclaimed (indicating memory leaks). Each of these hypotheses can be expressed as a relation (a measure relative to an event): 1) the number of requests *relative to* each time epoch, 2) the amount of memory allocated *relative to* each request, 3) the amount of memory reclaimed *relative to* each request. Relational debugging verifies the hypotheses by comparing the three relations in executions with and without the observed performance anomaly. In this example, 1) and 2) are the same, while 3) decreases significantly, suggesting memory leaks. Relational debugging further pinpoints the root cause of the memory leak by analyzing fine-grained relations. It finds that *relative to* all memory objects not reclaimed by the garbage collector (GC), many more are unreachable by pointers in the abnormal execution than the normal execution—a bug in the GC mistakenly treats constant values as pointers.

Relational debugging is highly effective with a minimal two executions (a good and a bad), eliminating the pain point of producing and labeling many different executions required by traditional statistical techniques [24, 26, 29, 32, 38, 39]. Notably, relational debugging utilizes the repetitiveness of performance symptoms which accumulate during the execution—a single execution offers a large sample of normal or abnormal patterns. Relational debugging is *generic* to performance problems with different types of root causes, including inefficient code, misconfigurations, and workload changes, etc. Moreover, relations can describe different types of symptoms such as slowdowns and memory overuse.

We realize relational debugging by developing a practical tool named Perspect. Perspect is fully *automatic*; it does not require manual instrumentations or annotations. Perspect takes the symptoms (such as a program counter that indicates excessive memory usage or a function with abnormal execution time) as inputs. It outputs the relations that are 1) *causally relevant* to the symptoms and 2) have significant impacts on the performance measures of the symptom; such relations describe the root causes of the performance problems. Perspect directly operates on x86 binaries to accommodate real-world diagnosis scenarios (e.g., when the binary build is nontrivial), and can tolerate small differences in the binaries.

Perspect focuses on capturing a small set of relations that can pinpoint the root cause. Instead of tracking all possible relations of every runtime event, Perspect reduces the search space by identifying runtime events that are causally related to the symptoms through control or data flow. Perspect then filters out relations that are not changed between the good and bad executions. For relations that are changed between

the executions, Perspect automatically differentiates between relation changes that reflect the *effect* (e.g., a decrease of reclaimed memory relative to each request), and changes that reflect the *cause* (e.g., an increase in objects not referred by real pointers). These strategies effectively filter out most of the irrelevant relations, with the remaining relations being root cause candidates. Lastly, Perspect ranks root-cause relations based on their impacts on performance measures of the observed symptom, and outputs them in descending order.

Perspect is carefully implemented so its analysis is both *precise* and *scalable* to real complex systems. It has an efficient algorithm that computes all the relations by traversing the dependency graph only once. In addition, it distributes the precise but expensive data-flow dependency analysis onto different servers. Finally, Perspect is able to handle the difference between two different versions of the binary executables.

We evaluate Perspect on twelve real-world performance issues from complex systems (Go runtime, MongoDB, Redis, and Coreutils), covering different symptoms (slowdown and memory overuse). Perspect effectively locates the root causes of these challenging issues. Notably, we applied Perspect to *two open issues* where developers failed to find the root causes; Perspect successfully located the root causes of both issues which are confirmed by the developers. For an issue where the root cause is located outside the target program (in the OS kernel), which took developers a long time to debug, Perspect correctly excluded the root cause from the application code, since it detects no significant relation changes.

In summary, this paper makes the following contributions:

- We present relational debugging, a new technique that analyzes the relations between causally related events, seizing the essence of performance debugging.
- We build Perspect, a practical tool that realizes relational debugging for large, complex real-world systems. Perspect directly operates on x86 binaries and accommodates real-world diagnosis/debugging scenarios.
- We show that Perspect can effectively locate the root causes of real performance problems, and can help resolve two previously unresolved issues. The source code of Perspect and the dataset are available at <https://gitlab.dsrg.utoronto.ca/dsrg/perspect>.

## 2 Relational Debugging by Examples

We use two real-world examples to show how relational debugging locates the root causes of challenging performance problems in complex software systems. Both problems are among the most challenging performance issues faced by developers, who were unable to locate the root causes with existing tools. Specifically, the Go runtime bug (§2.1) took a year of investigation, and the MongoDB bug (§2.2) is an open issue that developers failed to diagnose. Perspect automatically pinpoints the root causes in the form of relations.

## 2.1 Go-909: A Memory Leak

Go-909 is among the most famous performance bugs in the Go runtime. The developers reported that “*garbage collection is ineffective on 32-bit*” systems, causing workloads to run out of memory [2]. The same bug resulted in 9 other tickets (which turned out to have the same root cause) and at least 2 extensive discussion threads on Golang’s email list. The bug was also discussed in Hacker News with 147 comments [4].

### 2.1.1 Challenges of Debugging Go-909

Debugging Go-909 was very challenging not only for application developers but also for developers of the Go runtime. During the course, many wrong hypotheses, some of which were wildly off, were developed. For example, a developer believed that the bug was caused by the Go runtime forgetting to `munmap` freed memory [1]. There are at least three other bugs, of which developers could not agree on the root cause, that were eventually attributed to Go-909. After more than a year of investigations, the root cause was discovered through a trial-and-error process: the bug can be worked around by commenting out specific packages that contain a lot of static constants.

Existing performance debugging techniques can hardly address Go-909. First, Go-909 does not always cause a clear-cut out-of-memory error; in fact, many developers reported the bug simply after noticing their programs using more memory than expected [1, 3]. Moreover, since the root cause is not in program inputs, isolating faulty inputs using X-ray [15] or delta debugging [48] does not help. The root cause also can hardly be revealed by statistical debugging [32, 39], because it does not manifest in any abnormal predicates such as branch targets, unexpected return values, or scalar-pairs [39]. In fact, the memory leak also occurred in the reference executions (64-bit systems), only affecting many fewer objects.

### 2.1.2 Root Cause

Figure 1 shows the simplified code snippet in the buggy version of the Go runtime. Go programs invoke `runtime.malloc` to allocate memory and the Go runtime uses a mark-and-sweep garbage collector (GC). Once an object is allocated (L2), `runtime.malloc` increments the `heap_size` counter (L3).

The `mark` function looks for objects that are reachable through variables on the stack and in the data segments. Unmarked objects will later be reclaimed by `sweep`. During the stack scan, `mark` takes the pointer to the start of the stack and data segments (`b`), as well as the size of the respective regions (`n`). For every word on the stack and data segments, it initially assumes it to be a pointer and checks whether it points to an address inside the heap’s range (L15). If so, `mark` sets the “marked” bit in the metadata of the object (L18–19). Then, `mark` uses an iterative worklist `w` to further scan the memory based on the marked pointers. Later, `sweep` goes through each span, a memory region containing same-sized blocks. The

```
1 void* runtime.malloc(uintptr size, ...) {
2     void *p = runtime.Alloc(...);
3     heap_size += size;
4     uintptr bits = get_metadata(p);
5     ...
6     set_metadata(p, bits);
7     return p;
8 }
9 // Mark objects reached by pointers
10 static void mark(byte *b, int64 n) {
11     void **w = get_buffer_head();
12     while(b != nil) { ...
13         for(i = 0; i < n; i++) {
14             byte *p = (byte*)b[i];
15             if(p < HEAD_START || p >= HEAD_USED)
16                 continue;
17             uintptr bits = get_metadata(p);
18             bits |= BIT_MARKED; /* set mark bits */
19             set_metadata(p, bits);
20             *w++ = p;
21         }
22         b = *--w;
23         n = get_size(b);
24     }
25 }
26 // Reclaim unreachable objects
27 static void sweep(void) {
28     uintptr size = getsize(span);
29     for(byte *p = span->start; ... p += size) {
30         uintptr bits = get_metadata(p);
31         if((bits & BIT_MARKED) != 0) {
32             bits &= ~BIT_MARKED; /* clear mark bit */
33             continue;
34         }
35         set_metadata(p, bits);
36         runtime.Free(p, size, ...);
37         heap_size -= size;
38     }
39 }
```

GC log heap\_size is logged

GC log heap\_size is logged

```
./Perspect run_64 run_32
run_64: R<(L7|L18) = 0.99 // 64-bit (good run)
run_32: R<(L7|L18) = 0.01 // 32-bit (bad run)
```

Figure 1: Code snippets showing how Perspect locates the root causes of Go-909 by pinpointing the changed relation between L7 and L18 by comparing the two runs.

loop at L29 goes through each block, checks if the marked bit is set, and if so, clears the mark bit (L32) and continue on to the next block. Otherwise, it frees the object and decrements the heap size (L37).

The implementation of `mark` suffers from fake pointers—non-pointer variables that happen to have values within the range of `HEAP_START` and `HEAP_USED` (L15). The objects pointed to by those variables will not be reclaimed.<sup>1</sup> The defect affects both 32- and 64-bit systems; however, fake pointers occur orders of magnitude more frequently in 32-bit systems than 64-bit systems due to data layouts differences.

<sup>1</sup>This is a known side effect of using a conservative garbage collector.

### 2.1.3 Relational Debugging Go-909

Perspect takes as inputs a good run (which uses the 32-bit Go runtime) and a bad run (which uses the 64-bit Go runtime) of the Go program provided by the bug reporter, as well as the symptom. Since the bug manifests in abnormal heap sizes in the GC log, we (users) feed Perspect the `heap_size` variable which records the heap size value printed in the log. Perspect identifies the instructions that modify `heap_size`, i.e., L3 and L37 in the code of Figure 1, and Perspect treats these instructions as symptom instructions. Perspect will not only analyze what causes these symptom instructions to execute, but also what prevents these symptom instructions from executing; To do this, Perspect also identifies “negation” symptoms which are instructions that directly prevent a symptom instruction from executing, for example, L18, because each time L18 executes which marks and object, it directly prevents an instance of L37 which reclaims the object.

Perspect carries out relational debugging starting from instructions that directly determine the `heap_size` (L3, L37, and L18). It builds relations between symptom instructions and their causal predecessors. In this case, Perspect efficiently locates the root cause to a single relation (see Table 1 for notations):

$$R\blacktriangleleft(L7_{\text{malloc.return}} \mid L18_{\text{mark}})$$

On 64-bit systems, the relation is expected to be 1 : 1, indicating that for every marked object on L18, there exists a dependency on a pointer returned by `malloc`. Yet, on 32-bit systems, the relation drops to 1 : 0.01, i.e., only 1% of the marked objects have a pointer returned by `malloc`. The remainings are pointed to by fake pointers (constant values).

Note that the 1 : 1 relation in the reference run on 64-bit systems is *not* an invariant. Precisely, Perspect observed the relation to be 1 : 0.99, i.e., 99% of the marked objects are pointed to by a pointer returned by `malloc`. This is because the defect still exists in 64-bit systems, but only affecting 1% of the objects in the reference run.

$R\blacktriangleleft(L7 \mid L18)$  is not the only relation built by Perspect. Taking L18 as an example, Perspect builds four relations w.r.t L18’s causal predecessors L1 and L10:

- $R\blacktriangleleft(L1 \mid L18)$ : the distribution of the number of marked objects that depend on `malloc`;
- $R\blacktriangleright(L18 \mid L1)$ : the distribution of the number of times an object (still reachable by real pointers) gets marked;
- $R\blacktriangleleft(L10 \mid L18)$ : the distribution of the number of marked objects that depend on `mark`;
- $R\blacktriangleright(L18 \mid L10)$ : the distribution of the number of objects marked per `mark` call;

Perspect filters out  $R\blacktriangleright(L18 \mid L1)$  because the distribution of the lifetimes of objects reachable by real pointers do not change significantly between the good and bad run; and Perspect filters out  $R\blacktriangleleft(L10 \mid L18)$  because each marked object always depend on one invocation of `mark`.  $R\blacktriangleright(L18 \mid L10)$  is

$L_n$	An static instruction at line $n$
$eL_n_i$	The $i$ -th instance of $L_n$ in the execution
$S$	A symptom instruction
$eS_i$	A symptom event
$P$	A static insn. & causal predecessor of $S$
$P^+$	A static insn. & direct causal successor of $P$
$R\blacktriangleright(S \mid P)$	A forward relation between $P$ and $S$
$R\blacktriangleleft(P \mid S)$	A backward relation between $P$ and $S$
$R\blacktriangleleft\blacktriangleright(P, S)$	A pair of forward and backward relations
$R_?(P, S)$	A relation btw. $P$ and $S$ of unspecified direction

Table 1: Notations for relations.

changed across the runs, because fake pointers causes many more objects to be marked during each `mark` call in the bad run, but Perspect also excludes it because relational debugging recognizes that the relation only reflects the *effect* of the root cause, but is not the root cause. Finally, for  $R\blacktriangleleft(L1 \mid L18)$ , Perspect refines it to the most specific variant,  $R\blacktriangleleft(L7 \mid L18)$ . The other relations (e.g., those w.r.t symptom instructions at L3 and L37) are handled in similar ways, and eventually filtered out. We discuss Perspect’s filtering and refinement techniques in §3.3.

## 2.2 MongoDB-57221: A Slowdown

*“[Perspect’s result] ties all the pieces together into a nice explanation. That explanation being, having some unnecessary cursors simply open on failed plans isn’t strictly the problem. It’s that we’re paying the (also unnecessary) cost to reposition them after every delete + restore.”*

—MongoDB developer’s comment on Perspect’s result.

MongoDB-57221 is an open bug which developers were unable to diagnose. It is triggered by executing a query that deletes all the records in the table. The query could slow down by 5x on the buggy version. During the deletion, MongoDB uses a cursor, i.e., a pointer to a record in the table that indicates the current position, to locate each record. It advances the cursor to the next record after deleting the previous one; this process is known as cursor restoration.

This bug is caused by maintaining unnecessary cursors on multiple query plans. Before the query execution, MongoDB generates multiple query plans, performs a sandboxed trial of these plans, and chooses the best-performing plan. Different query plans use different indexes, thereby deleting records in different orders. The actual order of the deletion is determined by the index of the *winning* plan. However, MongoDB still keeps the rejected plans and their cursors. More importantly, it restores the cursor of each rejected plan following the same order as the winning plan. Whereas for the winning plan, restoring the cursor means simply moving to the next position, for the losing plan, restoring the cursor requires traversing through many already deleted records. And if the number of deleted records encountered exceeds a threshold, it flags the page for eviction. The increase in unnecessary evictions leads to the slowdown.

Developers were unable to understand the root cause of this bug, despite them quickly identifying evictions being the bottleneck based on profiling and being aware of the existence of multiple query plans. However, they could not explain *why* excessive evictions occurred, because they could not establish the causal link between evictions and the cursor restoration of the rejected plans. This led to rounds of ping-pongs between the Storage Engine team (responsible for the eviction) and the Query Execution team (responsible for maintaining multiple plans). The storage team suspected that the slowdown was caused by maintaining multiple plans, but the query execution developers believed that it was cheap to keep multiple plans around. And they further suspected that the slowdown was caused by the threshold misconfiguration that triggered the eviction. In the end, seven different developers actively discussed this issue for over a month. The JIRA discussion has over 3,000 words in 18 comments, with multiple rounds of reproduction and profiling with multiple screenshots posted. And the two teams had multiple off-line teleconference discussions. Still, they were unsure why the slowdown occurred.

Perspect pinpoints the root cause and explains the slowdown. It captures that the costly evictions are causally dependent on the restoration of multiple cursors. Figure 2 displays a simplified version of static dependency graph for eviction. Starting from *eviction* as the symptom, Perspect returns the root cause candidate: (1)  $R \blacktriangleright (\text{cursor\_search} \mid \text{restore})$ , where *restore* invokes *cursor\_search* once in the good run to restore one cursor, but twice in the bad run to restore two cursors. Perspect infers that restoring an additional cursor causes a significant increase in evictions in the bad run.

Moreover, Perspect specifically infers that during cursor restoration, additional traversals through dead records increased evictions. It returns (2)  $R \blacktriangleleft \blacktriangleright (\text{eviction} \mid \text{search\_forward})$  as a new pair of relations unique to the bad run: *search\_forward* is invoked by *cursor\_search* to search for the next cursor position by traversing forward in the records. In the good run, *search\_forward* almost always locates the next cursor position immediately, triggering no evictions; whereas in the bad run, *search\_forward* traverses through many dead records and triggers additional evictions. Perspect also returns (3)  $R \blacktriangleright (\text{search\_backward} \mid \text{cursor\_search})$  as a root cause candidate. In the good run, *cursor\_search* invokes *search\_backward* only 1% of the time, because *search\_forward* locates the next cursor position most of the time; however, in the bad run, *cursor\_search* invokes *search\_backward* half of the time. The increased searches lead to additional evictions.

### 3 Perspect

Generally speaking, debugging a performance problem takes three steps: 1) observing symptom(s), 2) capturing runtime events that are causally related to the symptom(s), and 3) locating the root cause. Perspect automates the last two steps, taking the symptoms as its inputs. Perspect supports different

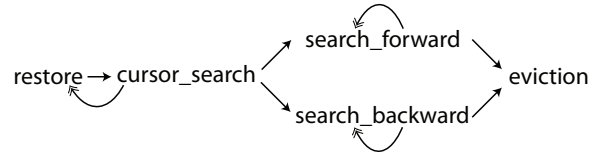


Figure 2: A simplified version of the static dependency graph for eviction. Each edge with a single arrow represents a dependency. An edge with a double arrow represents a backedge in a loop. *restore* loops through every cursor and restores each by invoking *cursor\_search*. *cursor\_search* then invokes *search\_forward* which looks for the next record by iterating forward. If *search\_forward* returns without locating the next record, *cursor\_search* will then invoke *search\_backward*. If *search\_backward* or *search\_forward* encounters too many dead records, it will trigger *eviction*.

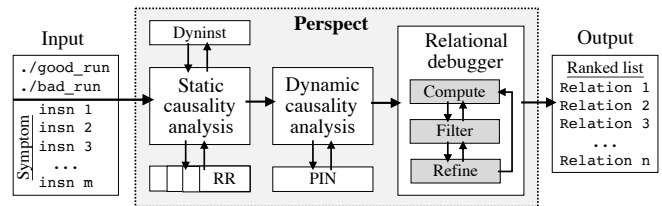


Figure 3: An overview of Perspect’s workflow

forms of symptoms, including: program variables that record the symptoms (e.g., *heap\_size* in Go-909), slow functions (such as *eviction* in MongoDB-57221), and basic blocks (captured by profilers like *gprof* [12]). Perspect automatically identifies the instructions related to the input symptoms as the starting points of its analysis (§3.1). Perspect outputs a list of relations that explain the root cause in descending order based on their impacts on the observed symptom.

Figure 3 shows the workflow of Perspect. Perspect uses causality analysis to reduce the search space of relational debugging to a small set of instructions and their runtime instances that are causally related to the symptom (see §3.2). Perspect then performs relational debugging to build relations with regard to the symptom. It filters out relations that are irrelevant to the symptom, refines relations to be specific to the root cause, and ranks relations based on their impacts on the observed symptom (see §3.3).

#### 3.1 Bootstrapping with Symptoms

Perspect bootstraps itself by identifying the instructions that reflect the observed symptoms. If the symptom is a performance counter recorded in a program variable (such as *heap\_size* in Go-909), Perspect identifies the instructions that use the variable as an operand. If the symptom is a function or a basic block (typically captured by profilers), Perspect identifies the first instruction of the function or the first instruction in the basic block. Hence, Perspect converts different types of symptom inputs to unified starting points in the form



of instructions, termed *symptom instructions*, denoted as  $\mathcal{S}$  in Table 1. A dynamic instance of the symptom instruction is a *symptom event*.

Each symptom instruction is assigned a *weight*. The weight can be the value of a variable in an instruction, such as `size` in L3 or L37 in Go-909 (Figure 1), or the estimated time or cycles taken by a code block (e.g., `eviction`).

Perspect also identifies instructions that prevent the occurrence of symptoms (i.e., the negation of a symptom) as a special type of symptom instructions. An example is L18 in Go-909. Perspect searches every conditional branch that dominates a symptom instruction, which could prevent the symptom from occurring (e.g., L31 in Go-909 w.r.t L37). It then identifies the instructions that determine the branch conditions (e.g., L18 in Go-909). In practice, we find it suffices to only include instructions of negation for the initial symptom instructions. Therefore, in our current implementation, Perspect does not recursively search for negation symptoms.

## 3.2 Causality Analysis

Perspect reduces the search space of relational debugging by restricting the subsequent analysis to a small subset of instructions and their runtime instances that are causally related to the symptoms. The high-level idea is to dynamically track instructions that the symptoms are causally dependent on through control- and data-flow (aka information flow) during the execution of the good or bad reproduction runs. Specifically, Perspect generates a *dynamic* program dependency graph that contains instances of instructions that the symptom is causally dependent on.

The causality tracking is done in two phases. Perspect first generates the *static* program dependency graph (SDG) [25] for all the symptom instructions from the program. In the SDG, a node  $v$  is an instruction and an edge  $(u, v)$  represents a causal dependence, either a data dependence (a data value  $v$  depends on) or a control dependence (a control condition on which  $v$  depends on). We call  $u$  a *causal predecessor* of  $v$  and  $v$  a *causal successor* of  $u$ . To generate the SDG, Perspect performs *backward* causality tracking: it starts from each symptom instruction (including negation symptoms) and recursively includes causal predecessor instructions by tracking control or data flow.

Perspect then automatically instruments the instructions in the program binary that belong to the SDG; it later generates *dynamic* program dependency graphs (DDGs) by running the program binary and monitoring the execution of each instrumented instruction. Different from the SDG, which consists of static instructions, in a DDG, a node is a runtime *event*—an instance of an instruction in the execution. Each instruction in the SDG can correspond to multiple events in a DDG. We use  $eLn_i$  to denote an event of the  $i$ -th occurrence of the instruction at line  $n$  (i.e.,  $Ln$ ) in the execution (see Table 1).

Section 4 describes the implementation details.

## 3.3 Relational Debugger

Within the scope of instructions that are causally related to the symptom(s), Perspect computes the relations between the symptom instructions and their causal predecessors in the SDG, based on runtime dependencies derived from the DDGs (§3.3.1). Perspect only considers relations that are changed between the good and the bad executions as potential root causes by filtering out unchanged relations (§3.3.2). Perspect further refines each relation until it finds the specific relation that captures a root cause of the change in the number of symptom events between the good and the bad executions (§3.3.3). The filtering and refinement steps are iterated repeatedly to select a minimal set of relations as the candidates of the root cause (Figure 3). Lastly, Perspect ranks the root-cause candidate relations based on their impact on the symptoms (§3.3.4).

We use Go-909 (Figure 1) as a running example when explaining the above components.

### 3.3.1 Computing Relations

For each symptom (including the negative symptoms), Perspect computes the relation between an instruction  $P$ , which the symptom depends on causally, and the corresponding symptom instruction  $\mathcal{S}$ . Both  $P$  and  $\mathcal{S}$  are nodes in the SDG generated in §3.2. The relation is computed based on the DDG (§3.2) which records runtime events of  $P$  and  $\mathcal{S}$  during the executions. Perspect computes relations for the good run and the bad run, respectively.

Perspect starts by only considering the relation between  $\mathcal{S}$  and the root nodes of the SDG as  $P$ . These root nodes are typically the entry point of a software module and the `main` function. It gradually considers other events on the causal dependency paths between the root node and  $\mathcal{S}$  using a refinement process described in §3.3.3.

Perspect computes both *forward relations* and *backward relations*. A forward relation is defined as  $R \blacktriangleright (\mathcal{S}|P) = \{n_i\}$ , where each element  $n_i$  in the set, which corresponds to an instance of instruction  $P$  (denoted as  $eP_i$ ) in the DDG, is the number of causally dependent  $\mathcal{S}$  instances ( $e\mathcal{S}_j, e\mathcal{S}_k \dots e\mathcal{S}_m$ ) of  $eP_i$ . Therefore, a relation can be viewed as a distribution; We use the mean of the distribution to represent a relation for simplicity. Here,  $P$  can be thought of as serving as a reference point, and  $\mathcal{S}$  as the object under observation.

For example, in Go-909, for the symptom instruction  $L18_{mark}$  (marking one object), Perspect constructs a relation  $R \blacktriangleright (L18_{mark}|L1_{malloc.start})$ , which represents the number of times each allocated object got marked. If the first allocated object gets marked (i.e., it results in an instance of  $L18$ ) but the second one does not, then  $R \blacktriangleright (L18_{mark}|L1_{malloc.start})$  would be  $\{1, 0\}$ . In practice,  $R \blacktriangleright (L18|L1)$  has a much larger sample size, because hundreds of objects are allocated and marked.

A backward relation is defined as  $R \blacktriangleleft (P|\mathcal{S}) = \{m_i\}$ , where each element  $m_i$ , which corresponds to an instance of  $\mathcal{S}$  in the DDG ( $e\mathcal{S}_i$ ), is the number of causally dependent  $P$  instances

( $eP_i, P_j \dots eP_k$ ) of  $eS_i$ . Opposite to a forward relation, for a backward relation, the symptom serves as the reference point, and the predecessor as the object under observation.

Regarding the example,  $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$ , which contains, for each marked object ( $L18_{mark}$ ), the number of causally connected `malloc` instances ( $L1_{malloc.start}$ ); each object pointed to by real pointers is connected to 1 instance of  $L1_{malloc.start}$  whereas an object pointed to only by fake pointers is connected to 0 instances.

Note that a backward and forward relation,  $R\blacktriangleleft(P|S)$  and  $R\blacktriangleright(S|P)$ , complement each other. A forward relation tells: “given the same unit of input, is the same number of symptom events produced?”, whereas a backward relation tells: “given the same symptom event, is it still produced by the same units of input?” In Go-909, fake pointers introduce additional causal paths through which the symptom at  $L18$  (marking one object) may occur. This is reflected in a change of the backward relation  $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$ , from 100% to 1% on average; the forward relation  $R\blacktriangleright(L18_{mark}|L1_{malloc.start})$ , reflecting the number of times each object (reachable from real pointers) gets marked, does not change significantly.

### 3.3.2 Filtering Unchanged Relations

Perspect filters out a relation  $R_{\gamma}(P, S)$  if it has not changed between the executions of the good and bad runs. Perspect determines if a relation has changed based on its distribution using the two-sample Kolmogorov-Smirnov test [27], with a confidence interval of 95%. For example, in Go-909, the relation  $R\blacktriangleright(L18_{mark}|L1_{malloc.start})$  does not change, because, for the objects still reachable from real pointers, the distribution of object life-spans (the number of times they get marked) does not change significantly; therefore, Perspect filters out this relation.

Furthermore, if a relation  $R_{\gamma}(P, S)$  is unchanged across two executions, it implies that the relations between any of  $P$ 's causal successors— $Q$ —and  $S$  have not changed. Perspect skips the computation of these relations. In other words, if there exists a causal successor  $Q$  where  $R_{\gamma}(Q, S)$  is changed, then  $R_{\gamma}(P, S)$  would be changed. Intuitively, it means that the same set of runtime events produces the same symptom events (forward relation) or the same set of symptom events is still produced by the same events (backward relation). This optimization allows us to skip many unnecessary relation computations.

In Go-909, Perspect filters out most of the relations at this step, and only keeps three relations (which will be further refined and filtered in §3.3.3):

- $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$ : the number of marked objects reachable from real pointers decreased;
- $R\blacktriangleright(L18_{mark}|L10_{mark.start})$ : the number of objects marked per `mark` call increased;
- $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$ : the number of objects reclaimed at  $L37$  per `sweep` call ( $L27$ ) decreased.

### 3.3.3 Relation Refinement

Perspect further refines the relations to replace a more “general” relation with a more “specific” one. Refinement is analogous to moving the reference point closer to the object under observation in relative motion. If a relation  $R_{\gamma}(P, S)$  is deemed *refinable*, Perspect replaces the relation with its child relations:  $R_{\gamma}(P_0^+, S), R_{\gamma}(P_1^+, S) \dots R_{\gamma}(P_n^+, S)$ , where  $\{P_0^+, P_1^+ \dots P_n^+\}$  are the direct causal successors of  $P$  (i.e., children of  $P$ ). Perspect iteratively refines a relation until it is no longer refinable or can be filtered out by §3.3.2.

Refinement aims to pinpoint the root cause(s). Without refinement, Perspect only outputs relations between  $S$  and root nodes  $R$  in the SDG, where  $R$  can be the entry point of a module or the `main` function. But the root cause(s) are often located at events on the causal paths connecting  $R$  and  $S$ . Intuitively, the root cause are events which, if executed, will inevitably cause the performance bug to manifest [50]. The refinement process aims to locate such events.<sup>2</sup>

We design the following two refinement rules:

Rule 1: A relation  $R_{\gamma}(P, S)$  is refinable, if there is no change in any of the relations between  $P$  and its children  $\{P_0^+, P_1^+ \dots P_n^+\}$ :  $R_{\gamma}(P, P_0^+)$ ,  $R_{\gamma}(P, P_1^+)$ , and  $R_{\gamma}(P, P_n^+)$ .

Intuitively, this rule says  $P$  is *not* a root cause; the root cause(s) is located further down the causal paths. Recall that the root cause(s) are events which, once executed, the performance bug will inevitably manifest. But now we have  $P+$  that occurred after  $P$  in *both* the good and bad run, and  $R_{\gamma}(P, P+)$  does not change. This means that after  $P$  executes, the performance bug may still be avoided when  $P+$  executes. So we should move one step forward on the causal chain to consider whether  $P+$  is the root cause.

With this rule, Perspect refines  $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$  to  $R\blacktriangleleft(L7_{malloc.return}|L18_{mark})$  in Go-909, because  $R\blacktriangleleft(L1|L7)$  is an invariant that does not change across executions. In the actual code, the program logic between  $L1$  and  $L7$  is complex; ruling out  $L1$  and narrowing it down to  $L7$  significantly helps the developer to understand the root cause.

Figure 4 further shows the sequence of refinements performed on  $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$ . Based on rule 1 we can refine it twice to  $R\blacktriangleright(L37|L31)$ , because neither  $R\blacktriangleright(L29|L27)$  nor  $R\blacktriangleright(L31|L29)$  changes.

Even if a relation is not deemed refinable by rule 1, we do not give up—it can still be refined based on rule 2:

Rule 2: Even if there is a change in  $R_{\gamma}(P, P_i^+)$ ,  $R_{\gamma}(P, S)$  is still refinable if the change in  $R_{\gamma}(P, P_i^+)$  is caused by the change of  $R_{\gamma}(P', P)$ , where  $P'$  is a predecessor of  $P+$  and  $P' \neq P$ .

Rule 2 differentiates whether a changed relation is a true root cause, or merely the *effect* (i.e., manifestation) of the root

<sup>2</sup>Zhang *et al.* defines the root cause as the *inflection point*: if we model the execution as a sequence of instructions, the inflection point in a failure execution  $F$  is the point of divergence with a non-failure execution  $N$  where  $N$  is the non-failure execution that has the longest common prefix with  $F$  [50]. Perspect’s refinement essentially locates such inflection points.

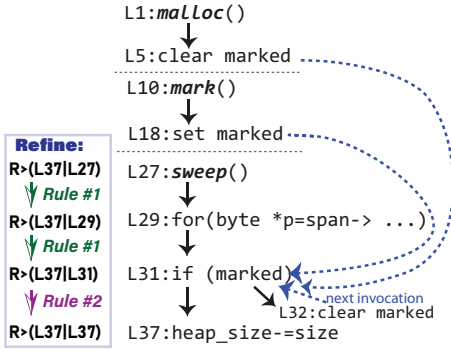


Figure 4: An example to illustrate the refinement rules on Go-909. On the right is (part of) the SDG; a solid edge indicates a control-flow dependency, whereas dotted edges represent dataflow dependencies.

cause. In the former case, we should not be able to find such a  $P'$ , whereas in the latter case, we can.

Specifically, we consider two cases in our implementation. The first is when  $P$  is a branch instruction,  $P^+$  is an instruction in the branch target, and  $P'$  is the dataflow direct predecessor of  $P$  that defines the branch condition variable. In this case, the change in  $R\blacktriangleright(P^+|P)$  is the effect of the change of  $R\blacktriangleleft(P'|P)$ , which affects the branch direction.

Consider  $R\blacktriangleright(L37_{sweep}|L31_{sweep.test})$ . Here,  $P$  and  $P^+$  are  $L31$  and  $L37$ , respectively. This relation decreased in the bad run since fewer objects are deemed reclaimable by  $L31$ , and is therefore no longer refinable according to Rule 1, as illustrated by Figure 4. However,  $L31$ 's direct dataflow predecessors include  $L18$ , which sets the mark bit ( $L18$  is the  $P'$  in this case). The decrease in  $R\blacktriangleright(L37|L31)$  is merely caused by the increase in  $R\blacktriangleleft(L18|L31)$ , i.e., more objects are being marked at  $L18$  before  $L31$  checks the marked bit. Therefore, according to Rule 2,  $R\blacktriangleright(L37|L31)$  is still refinable, and we refine it to  $R\blacktriangleright(L37|L37)$  (because  $L37$  is  $L31$ 's direct successor). It can be subsequently filtered based on §3.3.2 since a relation between two identical events doesn't change between runs. This is shown in Figure 4.

Note that we do not need to compute relations on this newly discovered  $P'$  separately, because our algorithm guarantees that this relation is computed through other causal paths from the root. For example, after Perspect found  $L18$  is the  $P'$  in the above example, it does not go on to compute relations between  $L18$  and its predecessors, because these relations are already computed through the causal path starting from `mark`.

The second case involves loops, when  $P^+$  is a loop head and  $P'$  is the loop tail. Consider  $R\blacktriangleright(L12_{mark.loop}|L10_{mark.start})$ . In this case,  $P$  is  $L10$  and  $P^+$  is  $L12$  (which is a loop head). This relation increased in the bad run because more objects are getting marked. However, this is caused by  $L12$ 's backedge from  $L24$  (loop tail, which is  $P'$ ) executing more often, i.e.,  $R\blacktriangleright(L24|L10)$  also increased by the same amount.

As a result, even though  $R\blacktriangleright(L12_{mark.loop}|L10_{mark.start})$  has changed,  $R\blacktriangleright(L18_{mark}|L10)$  can be further refined to  $R\blacktriangleright(L18|L12)$ . Eventually,  $R\blacktriangleright(L18|L12)$  will be filtered out because by further analyzing the dataflow predecessor of  $L12$  under Rule 2, Perspect finds that the number of times  $L12$  executes is controlled by the size of `w`, which in turn is dataflow-dependent on  $L18$  itself (i.e., each time an object is marked, it is pushed onto the queue `w` and popped from the queue later into `b` so `mark` can further scan the content of the object for more pointers). So the relation is refined to  $R\blacktriangleright(L18|L18)$  eventually.

By applying the two refinement rules iteratively, Perspect filters both  $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$  and  $R\blacktriangleright(L18_{mark}|L10_{mark.start})$ . Therefore, Perspect only reports one relation at the end of the filter-refine iterations:  $R\blacktriangleleft(L7_{malloc.return}|L18_{mark})$ .

### 3.3.4 Ranking Root-Cause Candidates

After the iterative compute-filter-refine process, the remaining relations are the ones that have not been filtered and are not refinable anymore. We call them root cause candidates.

Perspect ranks the root-cause candidates based on their estimated contributions to performance, in terms of the difference in performance relative to the predecessor  $P$ . Specifically, for a forward relation  $R\blacktriangleright(S|P) = \{n_i\}$ , where each  $n_i$  is the number of symptom instances that causally depend on  $eP_i$  (the  $i$ -th instance of  $P$ ), Perspect computes a weighted sum:  $\sum w_i \times n_i$ , where  $w_i$  is the average weight of the  $n_i$  symptom events;  $\sum w'_i \times n'_i$  is the weighted sum for the good run. Then the contribution to performance is estimated by  $\sum w_i \times n_i - (\sum w'_i \times n'_i) \times \frac{c_P}{c'_P}$ , where  $c_P$  and  $c'_P$  are the number of times  $P$  occurred in the bad and good run, respectively. Note that Perspect normalizes  $\sum w'_i \times n'_i$  with  $c_P/c'_P$  to obtain the performance relative to  $P$  in scenarios where the number of times  $P$  occurred has changed between the executions. (Say the change in  $P$ 's occurrences is caused by relation  $R\blacktriangleright(S|P')$ , where  $P'$  is a predecessor of  $P$ , the normalization helps correctly attribute performance impact between  $R\blacktriangleright(S|P)$  and  $R\blacktriangleright(S|P')$ .)

In a backward relation  $R\blacktriangleleft(P|S) = \{m_i\}$ , Perspect computes weighted sums:  $\sum w_i$ ,  $\sum w_j$  where  $w_i$  is the weight of the  $i$ -th instance of the symptom, and  $w_j$  is the weight of the  $j$ -th instance of  $P$  that can reach a symptom event; And  $\sum w'_i$ ,  $\sum w'_j$  are the weighted sums for the good run. Then the contribution to performance is estimated by  $\sum w_i - \sum w_j / (\sum w'_i / \sum w'_j)$ , where  $\sum w_j / (\sum w'_i / \sum w'_j)$  estimates the total number of symptom events, had the same number of symptom events been reachable from  $P$  instances in the good run; This formula also handles when the total number of reachable  $P$  instances from the symptom differs in the two executions. If the symptom has a negative polarity, as in the case of  $L37_{sweep}$ , which reduces the heap size as opposed to increasing it, Perspect multiplies its performance impact with  $-1$ .



## 4 Implementation

Perspect is implemented in 10,199 lines of C++ and 14,006 lines of Python. It is built on top of three tools, Dyninst [8] (a binary-level static analysis tool), RR [7] (a deterministic record-and-replay tool), and PIN [11] (a binary instrumentation tool). Perspect operates on application binaries directly.

A key challenge in our implementation is to scale Perspect to the real, complex systems software. This section describes a number of techniques we use for scalability.

### 4.1 Building Static Dependency Graph (SDG)

Perspect generates the SDG by recursively identifying instructions that are causal predecessors of the symptom instructions via control and data flow. (Figure 4 shows a snippet of the SDG on Go-909.) This is done by three components: 1) a *static analysis* (SA) process running Dyninst, 2) 64 *dynamic dataflow analysis* (DDA) processes running RR (across 4 servers), and 3) a controller. These components form a distributed system that parallelizes computation to scale to real-world systems.

The SA process iteratively infers the instructions on which the symptom instruction  $\mathcal{S}$  is control-flow dependent. It analyzes the control-flow graph provided by Dyninst, and only keeps those instructions that  $\mathcal{S}$  actually depends on. This analysis is first performed in the function ( $f$ ) that contains  $\mathcal{S}$ ; it is then repeated iteratively in the caller functions by tracing the call-sites starting from  $f$ .

To obtain dataflow dependencies, Perspect uses a combination of static and dynamic analysis. Perspect only uses Dyninst to obtain the dataflow dependencies of local variables stored in registers or on the stack with static offsets. On the other hand, when a variable is read from other memory locations, *i.e.* the heap or stack locations with non-static offsets, Perspect does not analyze them statically through pointer analysis, because precise pointer analysis can be hard to scale [31]. Instead, Perspect uses the DDA processes to dynamically identify such data dependencies in parallel.

For example, say  $\mathcal{S}$  is dominated by an if statement: `if (*p || *q)`; at this point, Perspect needs to infer the dataflow of both `*p` and `*q`, and Dyninst cannot infer the source of the dataflow precisely. Therefore, the SA process sends this request to the controller, which forwards it to a (pre-forked) DDA process to run the RR-guided reproduction. The DDA process first sets breakpoints at the `if` statement to determine the addresses of `*p` and `*q`. It then sets watchpoints at these two addresses and re-run the RR-guided reproduction.<sup>3</sup> (Since execution through RR is deterministic, addresses stay the same across multiple runs.) And via the watchpoints, Perspect locates the store instructions that defined `*p` and `*q`. The DDA process then sends these newly located store in-

<sup>3</sup> If a breakpoint or watchpoint is not hit in the RR-guided reproduction, Perspect will deem them causally irrelevant to the symptom events and ignore them.

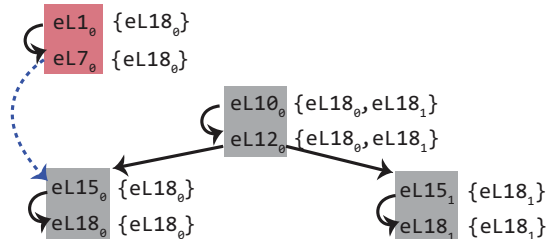


Figure 5: A simplified version of the Dynamic Dependency Graph (DDG) for the symptom instruction at L18 from Go-909. The *red* colour represents the `malloc` function, and the *grey* colour represents the `mark` function. Solid and dotted edges represent control and data flow. The set next to each event is the  $\mathcal{S}$ -set.

structions back to the SA process (via the controller). This causes the SA process to restart the analysis with these two instructions as the new starting points.

In practice, the SA is orders of magnitude faster than the DDA. Yet, the DDA can be parallelized: for example, the analysis of the dataflow source of `*p` and `*q` can be done in parallel. We create 64 DDA processes, each of which can set at most 4 watchpoints in each run (limited by the number of hardware watchpoints).

### 4.2 Building Relations

Once the SDG is obtained, Perspect instruments the program at each instruction in the SDG using PIN, and runs the instrumented program to obtain a trace of the good and the bad run, respectively. Perspect builds one DDG for each unique symptom instruction. Each vertex in the DDG is an event, and an edge is a control- or data-flow dependency. Figure 5 shows a simplified version of the DDG for the symptom instruction at L18 from Go-909. There are two objects in the DDG: The first one is reachable from a real-pointer, which means it's dependent on `malloc` (`eL10`, `eL70`), and the object gets marked (`eL100`, `eL120`, `eL150`, `eL180`). The second object is from a fake pointer; it also gets marked once in the same loop iteration as the first object (`eL100`, `eL120`, `eL151`, `eL181`), but has no dependencies on `malloc`.

Instead of traversing the DDG each time it needs to compute a relation, Perspect only carries out a one-pass traversal of the DDG to compute all the forward and backward relations. To compute forward relations, each node in the DDG keeps an  $\mathcal{S}$ -set, which is the set of all unique reachable symptom events. We initialize the  $\mathcal{S}$ -set of the symptom nodes to the symptom event itself. In Figure 5, `eL180` and `eL181`'s  $\mathcal{S}$ -sets are initialized with themselves. Perspect then traverses the DDG in *post-order* to iteratively compute the  $\mathcal{S}$ -sets. For each node  $N$ , its  $\mathcal{S}$ -set is the union of the  $\mathcal{S}$ -sets of all its children nodes. (Post-order traversal guarantees that  $N$ 's children are visited before  $N$ .) But keeping the  $\mathcal{S}$ -set of each node consumes too much memory. As an optimization, we replace



node  $N$ 's  $\mathcal{S}$ -set with its cardinality (i.e., number of elements or  $|\mathcal{S}\text{-set}|$ ) as soon as its  $\mathcal{S}$ -set is propagated to all of  $N$ 's parent nodes. For a forward relation  $R \blacktriangleright (\mathcal{S}|P) = \{n_i\}$ , each  $n_i$  is simply the  $|\mathcal{S}\text{-set}|$  of each event of  $P$ . For example, in Figure 5,  $R \blacktriangleright (L18|L10) = \{2\}$ , where 2 is the  $|\mathcal{S}\text{-set}|$  of  $eL10_0$ ; and  $R \blacktriangleright (L18|L15) = \{1, 1\}$ , where the two 1s come from the  $|\mathcal{S}\text{-set}|$  of  $eL18_0$  and  $eL18_1$ .

To compute backward relations, Perspect keeps a hashmap  $H$  for each symptom event. Each  $H$  keeps the number of reachable predecessor events for every corresponding predecessor instruction. For example, in Figure 5, Perspect keeps two  $H$ s, one for  $eL18_0$  and one for  $eL18_1$ . The  $H$  for  $eL18_0$  contains 5 entries:  $\{L15, L12, L10, L7, L1\}$ , and the count for each key is 1.  $H$  for  $eL18_1$  only contains 3 entries:  $\{L15, L12, L10\}$ , where the count for each key is also 1. A backward relation  $R \blacktriangleleft (L1|L18)$  is simply the set containing the count kept for  $L1$  in each  $H$ , which is  $\{1, 0\}$ .

**Optimization: two-phase analysis.** As an optimization, we perform our analysis in two phases. The first phase, or the “sketch” phase, only performs the analysis on the call graph. Specifically, each node in the SDG in this phase is a function, and each edge is a function invocation. The exceptions are functions that contain the symptom instructions: we directly connect the symptom instructions to the entry of these functions. We do not perform the expensive data-flow analysis in the sketch phase. Given this SDG, we build relations using the same algorithm: first obtain the DDG from the sketch SDG, and perform the relation analysis on this DDG. So, the  $P$  in the relations  $R_{\blacktriangleright}(P, \mathcal{S})$  we obtained is a function. For  $P$  whose relation changes, we zoom into  $P$  and perform the full data- and control-flow analysis described in §4.1. This optimization allows us to avoid the expensive dependency computations on functions that are not relevant to the root cause; it is particularly effective in large code bases like MongoDB where the symptom often has a deep call stack. In practice, this optimization reduces Perspect’s static analysis time by 10 times.

### 4.3 Handling Binary Difference

Perspect is able to compare relations generated from different binaries by matching each binary instruction to its corresponding one in the other binary, or between different binaries generated from the same source code (i.e., compiled for the 64- and 32-bit machines). Perspect first performs the source-level `diff` to establish the file and line number mapping between two versions. However, a line in the source code often compiles to multiple binary instructions, sometimes even multiple basic blocks of binary instructions. So we cannot only rely on source-level line number mapping to map binary instructions. Instead, for two binary instructions to be considered as the same between two version, they have to have 1) the mapping source-level line number, and 2) the same binary basic block number, assigned according to the postorder traversal of all the basic blocks of the same source code line, and 3) the same

offset within the basic block. If the instruction is not found at the same offset, Perspect also searches for nearby instructions.

## 5 Experimental Evaluation

Perspect’s premise is that relational debugging can automatically and effectively locate the root causes of real-world performance problems that are hard to diagnose by existing tools. We validate these hypotheses with three evaluation questions: 1) Can Perspect effectively locate the root cause of challenging performance problems? 2) Can Perspect’s output, in the form of relations, help users understand root causes? 3) What is the analysis time of Perspect?

- §5.1: Perspect effectively locates root causes of evaluated performance problems in Go runtime, MongoDB, Redis, and Coreutils. Perspect also correctly excludes a root cause from application code when it is in the OS kernel.
- §5.2: The output of Perspect, in the form of relations, can speed up debugging time by at least 10.87 times.
- §5.3: Perspect diagnoses 10/12 of the issues in 8 minutes on average, and diagnoses the other two in a few hours.

**Target applications and performance problems.** We evaluate Perspect on twelve real-world performance issues of four applications: the Go runtime, MongoDB, Redis, and Coreutils. All three are complex software systems, consisting of more than 220K, 6,955K, 37K, and 456K lines of code, respectively. The performance problems are collected from the issue trackers of the target applications, based on keywords like “performance”, “slow”, “degrade”, etc. Where possible, we focus on high-priority issues that cannot be simply answered by using a profiler but take significant human time and effort, as those are the problems that need advanced tools like Perspect.

We then try to reproduce these issues based on the steps described in the issue reports. Reproducing performance problems is nontrivial and time-consuming—many of the issues are imprecisely described (e.g., no version information or reproduction steps) and are hard to reproduce. In total, it took several person-months for us to prepare the dataset. We realize that our dataset has several “famous” bugs (e.g., Go-909 in §2) because they have more detailed information for reproduction.

As shown in Table 2, the twelve issues cover different symptoms and use cases. In terms of symptoms, nine caused slowdown; three caused memory overuse, including bloated heap size and resident set size (the amount of memory used by the process). There are three different types of performance baselines: five are from a different version, one from different hardware architecture, and the other five are from different inputs. Notably, we evaluated *two open issues* where developers were unable to diagnose them (MongoDB-56274 and -57221).

**Inputs.** Perspect takes as inputs of the reproduction of the performance problems. We directly used reproduction programs

	Issue	Description	Metric	Succ?	Rank	Abs. pred.?	Cand. relns.	SDG size	DDG size
Go runtime	909	Fake pointers stops GC from freeing dead objects	heap	Yes	1st	No	1	16054	516k
	7330	Performance of operator += is worse than single +	time	Yes	1st	Yes	1	26	200k
	8832	Hugepage promotion causes memory bloat	RSS	Partial	-	-	-	3140	6851
	11068	Printing is very slow for large Floats	time	Yes	1st	Yes	1	10650	1409k
	12228	More aggressive GC degrades performance	time	Partial	-	Yes	-	9886	39745
	13552	Not recycling large stack spans leaks memory	RSS	Yes	1st	No	1	18060	55580
Mongo	44991	Erroneous cache clear for common prefixed keys	time	Yes	1st	Yes	1	2109	461k
	<b>56274</b>	Slow when deleting opposite to search order	time	Yes	1st	No	3	56	6132
	<b>57221</b>	Slow due to moving cursor of obsolete query plan	time	Yes	1st	No	3	5100	268k
Redis	7595	performance downgrade after enabling TLS	time	Yes	1st	Yes	1	35	801
Core	930965	seq 84x slower with -equal-width	time	Yes	1st	Yes	1	668	20002
	1014738	du -exclude 4x slower when given a trivial string	time	Yes	1st	Yes	1	7563	20784

Table 2: **Perspect’s result on 12 real-world performance issues across 4 systems: Go runtime, MongoDB (“Mongo”), Redis, and Coreutils (“Core”).** Mongo-56274 and -57221 are two *open bugs*. “Metric” shows the type of performance metric that describes the symptoms. “Succ?” shows whether Perspect *successfully* locates the root cause. “Rank” shows the ranking of root-cause relations. “Abs. pred?” tells whether the root-cause relations break any absolute predicates. “Cand. relns.” shows the number of root-cause candidate relations. Where Perspect returns a pair of forward and backward relations, it is counted as one root cause candidate. “SDG size” and “DDG size” show the average SDG and DDG size from the good and bad runs, in terms of the number of instructions and their runtime instances, respectively.

attached in the reports, or created reproductions by closely following the descriptions in the reports. We find that except for Go-909, which provided three similar reproductions, all issues describe at most one good and one bad execution. Perspect is able to exploit high repetitiveness of runtime events within one execution, and works with two executions as is.

## 5.1 Effectiveness

Table 2 shows the effectiveness of Perspect in diagnosing the twelve performance bugs. The overall results are very positive. Perspect successfully locates the root causes for ten performance problems, and ranks the root-cause relation as the highest (or the only) suspect. Eight of them are closed issues and we use the criteria that the reported root cause has to be captured by the output relations of Perspect. For the two open bugs, the relations output by Perspect provided explanations of the root causes that were confirmed by the developers.

Perspect partially locates the root causes of the other two issues (Go-8832 and Go-12228). For Go-8832, Perspect correctly excludes the root cause (which lies in Linux) from the Go runtime. For Go-12228, the source codes changed significantly; Perspect is unable to map the relations across the executions. In this case, Perspect outputs the relations between the symptoms and causal predecessors so that a human developer can complete the rest of the debugging process.

As shown in Table 2, Perspect is able to effectively nail down a very small set of root-cause candidate relations. This is attributed to its iterative filtering (§3.3.2) and refinement (§3.3.3); Our experiments confirm that the relations between most events and their direct successors do not change across

executions. Perspect also filters out most causally related events with low contributions to the symptoms.

Note that 10/12 of the evaluated issues have no clear-cut failures—they are reported because the programs ran slower or consumed more memory than their respective baselines; the remaining two only *occasionally* result in out-of-memory errors (Go-909 and Go-13552). Hence, those issues can hardly be diagnosed by tools for functional failures. In at least four issues, the root causes do not manifest in any absolute predicate changes—the relations captured by Perspect show that the root causes exist in both executions, only their distributions differ. Lastly, as shown by the sizes of SDGs and DDGs, there are too many causally related instructions and runtime events—causality analysis alone can hardly pinpoint the root cause in code.

We discussed how Perspect locates the root causes of Go-909 and MongoDB-57221 in §2. We briefly present a few more.

**Mongodb-44991.** Mongodb-44991 is major performance regression introduced in v4.2.1 and took developers several days to diagnose. Figure 6 shows the simplified code containing the root cause. As a memory optimization, Mongodb stores key prefixes only once per page [5]; hence, it needs to decompress a key before evicting it back to disk. If the same key has been decompressed before, Mongodb copies the cached data directly (L4) to avoid building the key from scratch (L6). In v4.2.1, L11 was erroneously added, which clears the `size` variable, effectively invalidating cached data (L4).

Perspect takes the inputs of two executions from the good version (v4.0.13) and the buggy version (v4.2.1) as reported

```

1 void convert(void *key) {...
2   get_key_info(key, &data, &size);
3   if (... && size > 0)
4     memcpy(key, data, size); // fast path
5   else
6     build_key(entry, key); // slow path
7 }
8 void get_key_info(void *key, void *data, int *size) {
9   data = get_data(key);
10  ...
11  size = 0; — Invalidated the condition of using cached data (fast path);
12 } — Erroneously introduced in v4.2.1.

```

Figure 6: The root cause of MongoDB-44991 (used in §5.2).

in the issue. Perspect locates a pair of relations in the bad run:  $R \blacktriangleleft (L6, L11)$  and reports it as the highest-ranked root cause.

**Go-13552.** Developers noticed that the RSS slowly crept over to 1GB, even though the actual heap usage stayed below 4MB [6]. Diagnosing this bug took 5 days, and the developers eliminated several wrong guesses before nailing down the root cause. First, they had a hard time deciding whether the problem came from the heap or the stack. Once they focused on the stack, they further thought that the memory bloat was due to normal stack spans not being recycled fast enough. Finally, they found the root cause to be a special type of large stack spans which were not recycled at all.

Perspect ranks a relation  $R \blacktriangleright (\text{sysMmap}|\text{allocLarge})$  the highest, indicating that the increased mmap allocations are for large-sized stack spans. This connects the two essential pieces of information together to pinpoint the bug.

**MongoDB-56274 (open issue).** MongoDB-56274 is another open issue we diagnosed using Perspect, and the root cause has been confirmed by developers. The developers noticed that deleting records in descending order was twice as slow as in ascending order. MongoDB deletes records iteratively: after it deletes the record, it searches for the next record to delete. The `search` function has a hard-coded order: it always looks for the next record in ascending order first (`search_forward`); if no record is found, it searches backwards in descending order (`search_backward`). Hence, when the deletion order is the same as the search order, the next record is always found immediately; but, when the deletion order is the opposite, MongoDB traverses through many deleted records, then searches in the opposite direction, causing the slowdown.

Perspect locates the root cause to the hard-coded search order logic; In particular, it identifies three relations that increased significantly in the bad run: 1)  $R \blacktriangleright (\text{search_backward}|\text{search})$ : in the good run, `search_backward` is rarely invoked, as the next record is always immediately located by `search_forward`; 2)  $R \blacktriangleright (\text{prev\_record}|\text{search\_backward})$  and 3)  $R \blacktriangleright (\text{next\_record}|\text{search\_forward})$  indicates increased number of records traversed in both directions of search.

**Go-8832.** Developers observed unexpected memory bloat and mistakenly thought it was caused by bugs from Go’s GC code. In fact, the root cause was Linux’s promotion of huge pages in the background, which bloated the resident set size (RSS) since the distribution of the base 4KB pages was sparse. The developers spent a lot of time examining incorrect hypotheses about bugs in the GC logic, making it one of the most discussed Go performance issues.

While the current implementation of Perspect cannot analyze the OS kernel, it can help rule out wrongly suspected buggy behaviors of the Go runtime. Specifically, after comparing relations associated with the symptoms `mmap` and `munmap`, Perspect outputs no root cause candidate relations.

## 5.2 Usability

We evaluated the usability of Perspect with a controlled user study. We tested on 20 programmers (who are not co-author of this paper) who indicated extensive experience in debugging and GDB.

We used Go-909 and MongoDB-44991 in the study to represent resource issues and slowdowns. Each participant was asked to debug one case without any help and a different case with Perspect; so each bug has two controlled groups for comparison. For each participant, we first described the bugs and helped reproduce them. We chose one of the two cases randomly and asked the participant to diagnose it without Perspect; then for the second case, we introduced relational debugging and allowed them to use Perspect. We limited the debugging session to two hours for each bug (not including setup or reproduction time). If the time was exceeded, we considered the bug unsolved.

For Go-909, we considered a participant to have caught the root cause if they concluded that unreachable objects got marked and prevented reclamations. For MongoDB-44991, we used the criteria that the participant had to locate the instruction that clears the `size` variable erroneously (L11 in Figure 6).

Our results show that when using Perspect, participants concluded the root cause *at least 10.87 times faster* than when not using Perspect. With the help of Perspect, all participants successfully located the root causes of both issues, with an average of 10 minutes; much of the time was spent on navigating code and understanding instructions pointed to by the relations. Without Perspect, only 5/10 of the participants concluded the root causes within two hours, with an average of one hour and 47 minutes.

Interestingly, we observed that without Perspect many participants had *manual practices* like relational debugging: they printed out counters to compare occurrences of functions or instructions in the good and bad runs, and ruled out ones that did not change. However, we observed that such manual effort was neither rigorous nor systematic. For example, for Go-909, many participants examined if GC happened less often, but did not realize objects reclaimed per GC cycle changed.



We interviewed participants after the debugging session. The most overwhelming feedback is that the relation semantic is intuitive and easy to understand. One suggestion is to visualize SDGs and DDGs alongside the changed relations, which we consider implementing via GUI support.

### 5.3 Analysis Time

For the twelve performance issues, Perspect takes an average of under one hour to output the results. For 10/12 issues, Perspect finishes under 20 minutes, with an average of 8 minutes. The other two take an average of 5.3 hours. Most of the analysis time was spent on static and dynamic causality analysis (§3.2); the relational debugger (§3.3) takes a small fraction of the total analysis time.

Static causality analysis takes 24 minutes on average. It is bottlenecked by repeatedly invoking RR to build non-local dataflow dependencies (§4). The worst-case complexity of static causality analysis is  $O(n * m)$ , where  $n$  is the number of dynamic instructions executed during each RR run, and  $m$  is the number of static instructions that are causally related to the symptom instructions. The two-phase optimization described in §4.2 reduces  $m$  significantly. We can further speed up static analysis by adding more servers to parallelize the invocation of RR runs (§4.1).

Dynamic causality analysis is bottlenecked by running the instrumented program in PIN. It takes on average 35 minutes across the 12 issues (<20 minutes for 10/12 issues). Perspect effectively reduces the DDGs's sizes by sampling one symptom event out of  $N$ , while keeping a large number of symptom events to maintain statistical significance.

In comparison, the relational debugger only takes a small fraction of the total dynamic analysis time, typically a few minutes. Reducing the size of the DDG also effectively reduces the average complexity of the relational debugger, which has a worst-case complexity of  $O(p^2)$ , where  $p$  is the number of instructions executed that are causally relevant to the symptom events.

## 6 Discussion and Limitations

Relational debugging provides a new way of understanding performance problems. We find it generally applicable to many challenging performance problems that do not manifest via clear-cut predicates. Relational debugging assumes that the relations in the executions are statistically significant. It is possible that an execution is too short. On the other hand, our evaluation shows that the executions based on the reproduction steps documented in real-world issue reports are mostly sufficient—there are enough repetitive patterns for Perspect to be effective. It is straightforward to apply Perspect to multiple runs if one is too short.

Our current implementation of Perspect shares some limitations of its building blocks. Specifically, Perspect cannot debug performance problems that are non-deterministic (e.g., they depend on the scheduling and timing of events), because

Perspect uses deterministic replay (RR [7]) and its dynamic instrumentation could change the timing. Please note: this does *not* mean that Perspect cannot debug multi-threaded systems—all the evaluated systems (except Coreutils) are multi-threaded. In fact, it is reported that the vast majority (>90%) of real-world performance problems are deterministic [28].

Perspect currently only supports native code. We plan to implement relational debugging for applications in managed languages like Java. We believe the implementation can be built on the JVM Tool Interface. Perspect can be easily extended to handle additional language constructs like exception handling etc.<sup>4</sup> We will also explore how to apply relational debugging to performance problems of distributed systems by analyzing relations of distributed events. Perspect can be extended to support metrics such as P95 latency etc.<sup>5</sup>

## 7 Related Work

Performance debugging with Perspect takes three steps: 1) identifying symptoms, 2) causality analysis, and 3) relational debugging for automatically pinpointing root causes. We discuss related work based on the three components.

**Automatic performance debugging/diagnosis.** The closest related work (in terms of locating root causes in code) is [39], which applies statistical debugging [32] to performance problems. The essential idea of statistical debugging is to identify predicates that have strong correlations with the failure. However, as we have shown in this paper, it is fundamentally limited to performance problems that manifest via absolute predicates. Moreover, since statistical debugging in [39] does not take causality into consideration, many of the observed predicates could be irrelevant to the symptom; To compensate, it requires a large number of highly variable good and bad executions. Another related work is X-ray [15] which summarizes performance costs of runtime events and attributes them to input and configuration values w.r.t the symptom. Different from Perspect, X-ray is designed for end users (e.g., sysadmins) and does not target root causes in the code. X-ray uses differential performance summarization which identifies branches where execution paths diverge and reasons about the performance difference between the two branch outcomes. In this sense, it also focuses on divergence of predicates between executions.

There are tools for debugging special types of performance problems with predefined patterns, such as loops [35, 40, 44],

<sup>4</sup> When Perspect detects a symptom instruction is causally related to an exception handler, it can perform the analysis at instructions that can potentially throw an exception that is caught by this handler, treating these instructions as symptom instructions.

<sup>5</sup> Instead of calculating weighted sums, Perspect can perform the z-test on the weight of each symptom event against the distribution of weights of all symptom events (symptom events with a z-test score of 1.645 corresponds to the 95th percentile). The rank of each relation can be the number of causally related outlier symptom events.



memory leaks [41], and data locality [30]. Differently, Perspect is designed to be a general debugging/diagnosis tool.

**Automatic functional failure debugging/diagnosis.** Prior studies developed techniques to pinpoint the root causes of functional failures in code, based on invariant analysis [24, 26, 38], log analysis [50] and statistical debugging [32]. Perspect focuses primarily on performance problems which have very different characteristics from function failures.

**Causality analysis.** Perspect applies relational debugging to instructions and their runtime events that are causally related to the symptoms. Many advanced techniques have been developed for causality analysis [16, 22, 29, 34, 37, 42, 45–47, 49, 53]. Perspect can potentially use them to enhance its causality analysis (§3.2). For example, we can further accelerate the causality analysis, learning from failure sketching [29], REPT [22] and ER [53] that use Intel PT to efficiently trace causally dependent instructions and augment the trace with symbolic execution [18, 21]. Argus [42] developed a way to annotate causality graphs with strong and weak edges, which can prioritize relational analysis of Perspect. SherLog [45], lprof [52], and Pensieve [49] show that runtime logs can be used with static analysis to guide the reconstruction of causal paths.

Our work is complementary to causality analysis for distributed systems (many targeting performance problems [14, 17, 20, 34, 43, 51]). Relational debugging for distributed systems based on distributed causality is our future work (§6).

**Profilers.** Profilers [9, 10, 12, 13, 19, 23, 36] are important utilities for performance debugging. Advanced profilers like [23, 36] can effectively identify true bottlenecks. They provide effective inputs for Perspect to locate root causes.

## 8 Conclusion

Debugging performance problems is (still) among the most challenging, time-consuming tasks. We presented relational debugging as a new way of understanding performance problems and locating their root causes in the code. Our key insight is that the root causes of performance bugs can be generalized to changes in relations between fine-grained runtime events, and by using relations, we capture root causes of performance bug existing semantics (such as invariants or predicates etc.) fail to capture. We developed Perspect to automate relational debugging. Perspect takes a minimal of just two executions (a good and bad run), and pinpoints the root causes of complex real-world bugs to a small number of root cause relations using an effective “filter-and-refine” algorithm. We further demonstrate Perspect’s effectiveness by diagnosing two open issues which developers were unable to diagnose using existing tools. Finally, we deploy a number of carefully designed optimizations to scale Perspect to large-scale code-bases. We open-sourced Perspect and will continue improving it towards a common toolkit for performance debugging.

## Acknowledgement

We thank our shepherd, Jason Flinn, and the anonymous reviewers for their feedback and comments on our work. We also thank Serguei Makarov for the suggestion to output binary instead of plain-text PIN logs for optimized performance. This work was supported by the Canada Research Chair fund, an NSERC Discovery grant, an NSERC Alliance Mission grant, and an NSF grant CNS-2130560.

## References

- [1] Go-1091: runtime: gob leaks memory for larger objects (above MMAP\_THRESHOLD?). <https://github.com/golang/go/issues/1091>, Sept. 2010.
- [2] Go-909: runtime: garbage collection ineffective on 32-bit. <https://github.com/golang/go/issues/909>, July 2010.
- [3] memory leak on 8g. <https://github.com/golang/go/issues/1210>, Oct. 2010.
- [4] Go: Severe memory problems on 32bit Linux. <https://news.ycombinator.com/item?id=3805302>, 2012.
- [5] File formats and compression. [http://source.wiredtiger.com/2.3.0/file\\_formats.html](http://source.wiredtiger.com/2.3.0/file_formats.html), 2014.
- [6] Go-13552: runtime: RSS creeps over 1GB even though heap is 4MB. <https://github.com/golang/go/issues/13552>, 2015.
- [7] rr: lightweight recording and deterministic debugging. <https://rr-project.org/>, 2017.
- [8] Paradyn/Dyninst - Welcome | Putting the Performance in High Performance Computing. <https://www.dyninst.org/>, 2021.
- [9] perf: Linux profiling with performance counters. [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page), 2021.
- [10] SystemTap. <https://sourceware.org/systemtap/>, 2021.
- [11] Pin: A Dynamic Binary Instrumentation Tool. <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>, 2022.
- [12] The GNU Profiler. [https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html\\_mono/gprof.html](https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html), 2022.
- [13] Valgrind: a memory profiling and debugging tool. <https://valgrind.org/>, 2022.
- [14] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP’03)* (Oct. 2003).
- [15] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI’12)* (Oct. 2012).

- [16] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Oct. 2010).
- [17] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).
- [18] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (Dec. 2008).
- [19] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC'04)* (June 2004).
- [20] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)* (Aug. 2016).
- [21] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)* (Mar. 2011).
- [22] CUI, W., GE, X., KASIKCI, B., NIU, B., SHARMA, U., WANG, R., AND YUN, I. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).
- [23] CURTSINGER, C., AND BERGER, E. D. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).
- [24] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)* (May 1999).
- [25] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349.
- [26] HANGAL, S., AND LAM, M. S. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'02)* (May 2002).
- [27] HODGES, J. J. The significance probability of the smirnov two-sample test. *Arkiv fiur Matematik*, 3 (1958), 469–486.
- [28] JIN, G., SONG, L., SHI, X., SCHERPELZ, J., AND LU, S. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)* (June 2012).
- [29] KASIKCI, B., SCHUBERT, B., PEREIRA, C., POKAM, G., AND CANDEA, G. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).
- [30] KHAN, T. A., NEAL, I., POKAM, G., MOZAFARI, B., AND KASIKCI, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).
- [31] LI, Y., TAN, T., MØLLER, A., AND SMARAGDAKIS, Y. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)* (Nov. 2018).
- [32] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (June 2005).
- [33] LINDEN, G. Marissa Mayer at Web 2.0. <http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html>, Nov. 2017.
- [34] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).
- [35] NISTOR, A., SONG, L., MARINOV, D., AND LU, S. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)* (May 2013).
- [36] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)* (May 2015).
- [37] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).
- [38] SAHOO, S. K., CRISWELL, J., GEIGLE, C., AND ADVE, V. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)* (Mar. 2013).
- [39] SONG, L., AND LU, S. Statistical Debugging for Real-World Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)* (Oct. 2014).
- [40] SONG, L., AND LU, S. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)* (May 2017).
- [41] VILK, J., AND BERGER, E. D. BLeak: Automatically Debugging Memory Leaks in Web Applications. In *Proceedings of*

*the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)* (June 2018).

- [42] WENG, L., HUANG, P., NIEH, J., AND YANG, J. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)* (July 2021).
- [43] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)* (Oct. 2014).
- [44] XIAO, X., HAN, S., ZHANG, D., AND XIE, T. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13)* (July 2013).
- [45] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XV)* (March 2010).
- [46] ZAMFIR, C., AND CANDEA, G. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys'10)* (Apr. 2012).
- [47] ZAMFIR, C., KASIKCI, B., KINDER, J., BUGNION, E., AND CANDEA, G. Automated Debugging for Arbitrarily Long Executions. In *Proceedings of the 14th Workshop on Operating Systems (HotOS-XIV)* (May 2013).
- [48] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200.
- [49] ZHANG, Y., MAKAROV, S., REN, X., LION, D., AND YUAN, D. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).
- [50] ZHANG, Y., RODRIGUES, K., LUO, Y., STUMM, M., AND YUAN, D. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).
- [51] ZHAO, X., RODRIGUES, K., LUO, Y., YUAN, D., AND STUMM, M. Non-intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).
- [52] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).
- [53] ZUO, G., MA, J., QUINN, A., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)* (June 2021).



# Accountable authentication with privacy protection: The Larch system for universal login

Emma Dauterman  
UC Berkeley

Danny Lin  
Woodinville High School

Henry Corrigan-Gibbs  
MIT

David Mazières  
Stanford

**Abstract.** Credential compromise is hard to detect and hard to mitigate. To address this problem, we present larch, an accountable authentication framework with strong security and privacy properties. Larch protects user privacy while ensuring that the larch log server correctly records every authentication. Specifically, an attacker who compromises a user’s device cannot authenticate without creating evidence in the log, and the log cannot learn which web service (relying party) the user is authenticating to. To enable fast adoption, larch is backwards-compatible with relying parties that support FIDO2, TOTP, and password-based login. Furthermore, larch does not degrade the security and privacy a user already expects: the log server cannot authenticate on behalf of a user, and larch does not allow relying parties to link a user across accounts. We implement larch for FIDO2, TOTP, and password-based login. Given a client with four cores and a log server with eight cores, an authentication with larch takes 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP).

## 1 Introduction

Account security is a perennial weak link in computer systems. Even well-engineered systems with few bugs become vulnerable once human users are involved. With poorly engineered or configured systems, account compromise is often the first of several cascading failures. In general, 82% of data breaches involve a human element, with the most common methods including use of stolen credentials (40%) and phishing (20%) [79].

When users and administrators identify stolen credentials, it is challenging to determine the extent of the damage. Not knowing what an attacker accessed can lead to either inadequate or overly extensive recovery. LastPass suffered a breach in November 2022 because they didn’t fully recover from a compromise the previous August [72]. Conversely, Okta feared 366 organizations might have been accessed when an attacker gained remote desktop access at one of their vendors. It took a three-month investigation to determine that, in fact, only two organizations, not 366, had really been victims of the breach [32].

Single sign-on schemes, such as OpenID [74] and “Sign in with Google,” can keep an authentication log and thereby determine the extent of a credential compromise. However, these centralized systems represent a security and privacy risk: they give a third party access to all of a user’s accounts and to a trace of their authentication activity.

An ideal solution would give the benefits of universal authentication logging without the security and privacy drawbacks of single-sign-on systems. For security, the logging service shouldn’t be able to authenticate on behalf of a user. For privacy, the logging service should learn no information about a user’s authentication history: the log service should not even learn if the user is authenticating to the same web service twice or to two separate web services.

In this paper, we propose larch (“login archive”), an accountable authentication framework with strong security and privacy properties. Authentication takes place between a user and a service, which we call the *relying party*. In larch, we add a third party: a user-chosen larch log service. The larch log service provides the user with a complete, comprehensive history of her authentication activity, which helps users detect and recover from compromises. Once an account is registered with larch, even an attacker who controls the user’s client cannot authenticate to the account without the larch log service storing a record that allows the user to recover the time and relying-party name.

The key challenge in larch is allowing the log service to maintain a complete authentication history *without becoming a single point of security or privacy failure*. A malicious larch log service cannot access users’ accounts and learns no information about users’ authentication histories. Only users can decrypt their own log records.

Larch works with any relying party that supports one of three standard user authentication schemes: FIDO2 [36] (popularized by Yubikeys and Passkeys [3]), TOTP [68] (popularized by Google Authenticator), and password-based login. FIDO2 is the most secure but least widely deployed of the three options.

A larch deployment consists of two components: a browser add-on, which manages the user’s authentication secrets, and one or more larch log services, which store authentication logs on behalf of a set of users. At a high level, larch provides four operations. (1) Upon deciding to use larch, a user performs a one-time *enrollment* with a log service. (2) For each account to use with larch, the user runs *registration*. To relying parties, registration looks like adding a FIDO2 security key, adding an authenticator app, or setting a password. (3) The user then performs *authentication* with larch as necessary to access registered accounts. Finally, (4) at any point the user can *audit* login activity by downloading and decrypting the complete history of authentication events to all accounts. The client can use auditing for intrusion detection or to evaluate the extent of the damage after a client has been compromised.



All authentication mechanisms require generating an authentication credential based on some secret. In FIDO2, the secret is a signature key and the credential is a digital signature; the signed payload depends on the name of the relying party and a fresh challenge, preventing both phishing and credential reuse. With TOTP, the secret is an HMAC key and the credential an HMAC of the current time, which prevents credential reuse in the future. With passwords, the credential is simply the password, which has the disadvantage that it can be reused once a malicious client obtains it.

Larch splits the authentication secret between the client and log service so that both parties must participate in authentication. We introduce split-secret authentication protocols for FIDO2, TOTP, and password-based login. At the end of each protocol, the log service holds an encrypted authentication log record and the client holds a credential. Larch ensures that if the client obtains a valid credential, the log service also obtains a well-formed log record, even if the client is compromised and behaves maliciously. At the same time, the log service learns no information about the relying parties that the user authenticates to.

We design larch to achieve the following (informal) security and privacy goals:

- *Log enforcement against a malicious client:* An attacker that compromises a client cannot authenticate to an account that the client created before compromise without the log obtaining a well-formed, encrypted log record.
- *Client privacy and security against a malicious log:* A malicious log service cannot authenticate to the user's accounts or learn any information about the relying parties to which the user has authenticated, including whether two authentications are for the same account or different accounts.
- *Client privacy against a malicious relying party:* Colluding malicious relying parties cannot link a user across accounts.

Larch's FIDO2 protocol uses zero-knowledge proofs [43] to convince the log that an encrypted authentication log record generated by the client is well-formed relative to the digest of a FIDO2 payload. If it is, the client and log service sign the digest with a new, lightweight two-party ECDSA signing protocol tailored to our setting. For TOTP, larch executes an authentication circuit using an existing garbled-circuit-based multiparty computation protocol [87, 84]. For password-based login, the client privately swaps a ciphertext encrypting the relying party's identity for the log's share of the corresponding password using a discrete-log-based protocol [46].

In the event that a user's device is compromised, a user can revoke access to all accounts—even accounts she may have forgotten about—by interacting only with the log service. At the same time, involving the log service in every authentication could pose a reliability risk (just as relying on OpenID does). We show how to split trust across multiple

log service providers to strengthen availability guarantees, making larch strictly better than OpenID for all three of security, privacy, and availability.

We expect users to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of TOTP authentications. Given a client with four cores and a log server with eight cores, an authentication with larch takes 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP). One authentication requires 1.73MiB of communication for FIDO2, 65.2MiB for TOTP, and 3.25KiB for passwords. TOTP communication costs are comparatively high because we use garbled circuits [84]; however, all but 202KiB of the communication can be moved into a preprocessing step.

Larch shows that it is possible to achieve privacy-preserving authentication logging that is backwards compatible with existing standards. Moreover, larch provides new paths for FIDO2 adoption, as larch users can authenticate using FIDO2 without dedicated hardware tokens, which could motivate more relying parties to deploy FIDO2. Users who do own hardware tokens can use them to authenticate to the larch log service, providing strong security guarantees for relying parties that do not yet support FIDO2 (albeit without the anti-phishing protection). We also suggest small changes to the FIDO standard that would substantially reduce the overheads of larch while providing the same security and privacy properties.

## 2 Design overview

We now give an overview of larch.

### 2.1 Entities

A larch deployment involves the following entities:

**Users.** We envision a deployment with millions of users, each of which has hundreds of accounts at different online services—shopping websites, financial institutions, news sites, and so on. Each user has an account at a larch log service, secured by a strong, unique password and optionally (but ideally) strong second-factor authentication such as a FIDO2 hardware security key. (In Section 6, we describe how a user can create accounts with multiple log services in order to protect against faulty logs.) A user also has a set of devices (e.g. laptop, phone, tablet) running larch client software and storing larch secrets, including cryptographic keys and passwords.

**Relying parties.** A relying party is any website that a user authenticates to (e.g., a shopping website or bank). Larch is compatible with any relying party that supports authentication via FIDO2 (U2F) [36, 80], time-based one-time passwords (TOTP) [68], or standard passwords. The strength of larch's security guarantees depends on the strength of the underlying authentication method.

**Log service.** Whenever the user authenticates to a relying party, the client must communicate with the log service. We envision a major service provider (e.g. Google or Apple) deploying this service on behalf of their customers. The log service:

- keeps an encrypted record of the user’s authentication history, but
- learns no information about which relying party the user authenticates to.

At any time, a client can fetch this authentication record from the log service and decrypt it to see the user’s authentication history. That is, if an attacker compromises one of Alice’s devices and authenticates to `github.com` as Alice, the attacker will leave an indelible trace of this authentication in the larch log. At the same time, to protect Alice’s privacy, the log service learns no information about which relying parties Alice has authenticated to. A production log service should consist of multiple, georeplicated servers to ensure high availability.

## 2.2 Protocol flow

**Background.** We use two-out-of-two *additive secret sharing* [75]: to secret-share a value  $x \in \{0, \dots, p-1\}$ , choose random values  $x_1, x_2 \in \{0, \dots, p-1\}$  such that  $x_1 + x_2 = x \pmod p$ . Neither  $x_1$  nor  $x_2$  individually reveals any information about  $x$ . We also use a cryptographic *commitment scheme*: to commit to a value  $x \in \{0, 1\}^*$ , choose a random value  $r \in \{0, 1\}^{256}$  (the commitment *opening*) and output the hash of  $(x||r)$  using a cryptographic hash function such as SHA-256. For computationally bounded parties, the commitment reveals no information about  $x$ , but makes it impractical to convince another party that the commitment opens to a value  $x' \neq x$ .

The client’s interaction with the log service consists of four operations.

**Step 1: Enrollment with a log service.** To use larch, a user must first *enroll* with a larch log service by creating an account. In addition to configuring traditional account authentication (i.e., setting a password and optionally registering FIDO2 keys), the user’s client generates a secret *archive key* for each authentication method supported. For FIDO2 and TOTP, the archive key is a symmetric encryption key, and the client sends the log service a commitment to this key. For passwords, the archive key is an ElGamal private encryption key, so the client sends the log service the corresponding public key. The client subsequently encrypts log records using these archive keys, while the log service verifies these log records are well-formed using the corresponding commitment or public key.

**Step 2: Registration with relying parties.** After the user has enrolled with a log service, she can create accounts at relying parties (e.g., `github.com`) using larch-protected credentials. We call this process *registration*. Registration works differently depending on which authentication mechanism the relying party uses: FIDO2 public-key authentication, TOTP

codes, or standard passwords. All generally follow the same pattern where at the conclusion of the registration protocol:

- the log service holds an encryption of the relying party’s identity under a key that only the client knows,
- the log service and client jointly hold the account’s authentication secret using two-out-of-two *secret sharing* [75],
- the relying party is unaware of larch and holds the usual information necessary to verify account access: an ECDSA public key (for FIDO2), an HMAC secret key (for TOTP), or a password hash (for password-based login), and
- the log service learns nothing about the identity of the relying party.

By splitting the user’s authentication secret between the client and the log, we ensure that the log service participates in all of the user’s authentication attempts, which allows the log service to guarantee that every authentication attempt is correctly logged.

The underlying authentication mechanisms (FIDO2, TOTP, and password-based login) only provide security for a given relying party if the user’s device was uncompromised at the time of registration; larch provides the same guarantees.

**Step 3: Authentication to a relying party.** Registering with a relying party lets the user later authenticate to that relying party (Figure 1). At the conclusion of an authentication operation, larch must ensure that:

- authentication succeeds at the relying party,
- the log service holds a record of the authentication attempt that *includes the name of the relying party*, encrypted under the archive key known only to the client, and
- the log service learns *no information* about the identity of the relying party involved.

The technical challenge here is guaranteeing that a compromised client cannot successfully authenticate to a relying party without creating a valid log record. In particular, the log service must verify that the log record contains a valid encryption of the relying party’s name under the archive key *without* learning anything about the relying party’s identity.

To achieve these goals, we design *split-secret authentication protocols* that allow the client and log to use their split authentication secrets to jointly produce an authentication credential. Our split-secret authentication protocols are essentially special-purpose two-party computation protocols [88]. In a two-party computation, each party holds a secret input, and the protocol allows the parties to jointly compute a function on their inputs while keeping each party’s input secret from the other. Our split-secret authentication protocols follow a general pattern, although the specifics depend on the underlying authentication mechanism in use (FIDO2, TOTP, or password-based login):

- The client algorithm takes as input the identity of the relying party, the client’s share of the corresponding authentication secret, the archive key, and the opening for the log service’s commitment to the archive key.

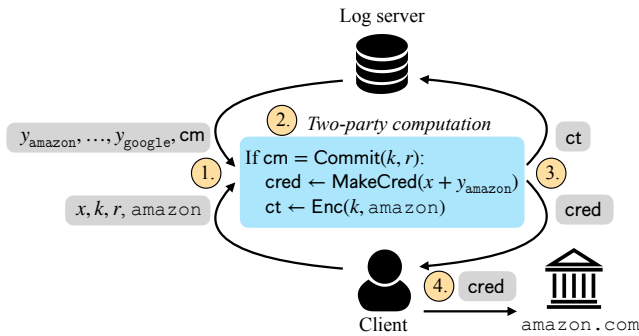


Figure 1: The client and log service run split-secret authentication where the client obtains the credential for `amazon.com` and the log service obtains an encryption of `amazon.com` under the client’s key. The client’s inputs are its share  $x$  of the authentication secret, the archive key  $k$ , a random nonce  $r$ , and the string `amazon.com`. The log’s inputs are its shares  $y_{amazon}, \dots, y_{google}$  of all the client’s authentication secrets and the commitment  $cm$  to the archive key generated at enrollment. The `MakeCred` function takes extra inputs for FIDO2 and TOTP.

- The log algorithm takes as input its shares of authentication secrets and the client’s commitment to the archive key (which it received at enrollment).
- The client algorithm outputs an authentication credential: a signature (for FIDO2), an HMAC code (for TOTP), or a password (for password-based login).
- The log algorithm outputs an encryption of the relying party identifier under the archive key.

In this way, the client and log service jointly generate authentication credentials while guaranteeing that every successful authentication is correctly logged. The client and log do not learn any information beyond the outputs of the computation. We use this general pattern to construct split-secret authentication protocols for FIDO2 (Section 3), TOTP (Section 4), and password-based login (Section 5).

**Step 4: Auditing with the log.** Finally, at any time, the user can ask the log service for its collection of log entries encrypted under the archive key. A user could do this when she suspects that an attacker has compromised her credentials. The user’s client could also perform this auditing in the background and notify the user if it ever detects anomalous behavior. The client uses the encryption key it generated during enrollment to decrypt log entries.

### 2.3 System goals

We now describe the security goals of larch (Figure 2).

**Goal 1: Log enforcement against a malicious client.** Say that an honest client enrolls with an honest log service and then registers with a set of relying parties. Later on, an attacker compromises the client’s secrets (e.g., by compromising one of the user’s devices and causing it to behave maliciously). Every

successful authentication attempt that the attacker makes using credentials managed by larch will appear in the client’s authentication log stored at the larch log service. Furthermore, the honest client can decrypt these log entries using its secret key.

**Goal 2: Client privacy and security against a malicious log.** Even if the log service deviates arbitrarily from the prescribed protocol, it learns no information about (a) the client’s authentication secrets (meaning that the log service cannot authenticate on behalf of the client) or (b) which relying parties a client has interacted with.

**Goal 3: Client privacy against a malicious relying party.** A set of colluding malicious relying parties learn no information about which registered accounts belong to the same client. That is, relying parties cannot link a client across multiple relying parties using information they learn during registration or authentication.

To be usable in practice, larch should additionally achieve the following functionality goal:

**Goal 4: No changes to the relying party.** Relying parties that support FIDO2 (U2F), TOTP, or password authentication do not need to be aware of larch. Clients can unilaterally register authentication credentials such that all future authentications are logged in larch.

### 2.4 Non-goals and extensions

*Availability against a compromised log service.* Larch does not provide availability if the log service refuses to provide service. We discuss defenses against availability attacks in Section 6.

*Privacy against colluding log and relying party.* If the log service colludes with a relying party, they can always use timing information to map log entries to authentication requests. Therefore, larch makes no effort to obscure the relationship between private messages seen by the two parties and only guarantees privacy when the relying party and log service do not collude.

*Limitations of underlying authentication schemes.* Larch provides security guarantees that match the security of the underlying authentication schemes. FIDO2 provides the strongest security, followed by TOTP, and then followed by passwords. For TOTP and password-based login, larch provides no protection against *credential breaches*: if an attacker steals users’ authentication secrets (MAC keys or passwords) from the relying party, the attacker can use those secrets to authenticate without those authentications appearing in the log. FIDO2 defends against credential breaches because the relying party only ever sees the client’s public key.

Larch does protect against *device compromise* for all three authentication mechanisms: even if an attacker gains control of a user’s device, generating any of the user’s larch-protected credentials requires communicating with the log service and results in an archived log record. If the user discovers the device break-in later on, she can recover from the log a list

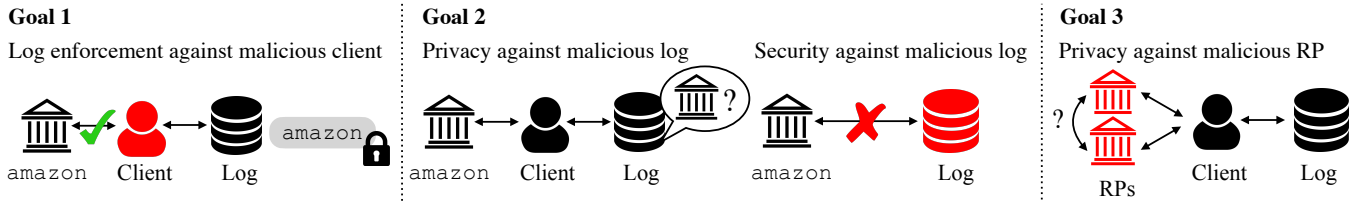


Figure 2: Larch security goals.

of authentications and take steps to remediate the effects of compromise (contacting the affected relying parties, etc.).

An attacker who compromises an account can often disable two-factor authentication or add its own credentials to a compromised account. Therefore, only an attacker’s first successful access to a given relying party is guaranteed to be archived in larch. That said, many relying parties send out notifications, require step-up authentication, or revoke access to logged in clients on credential updates, all of which could complicate an attack or alert legitimate users to a problem. Hence, it is valuable to ensure that all accesses with the original account credentials are logged. Larch can make this guarantee for FIDO2, where every authentication requires a unique two-party signature. It does not provide this guarantee with passwords, as the attacker learns the password as part of the authentication process: only the attacker’s first authentication to a given relying party will be logged. With TOTP, each generated code produces a larch log record. Some relying parties implement a TOTP replay cache, in which case one code allows one login. Other relying parties allow a single TOTP code to be used for arbitrarily many authentications in a short time period (generally about a minute).

Fortunately, when recovering from compromise, a user is most interested in learning whether an attacker has accessed an account zero times or more than zero times. For larch-generated credentials, users will always be able to learn this information from the larch log. However, if users import passwords that are not unique into larch, this guarantee does not hold. By default, the larch client software generates a unique random password for every relying party, but it also allows user to import existing legacy passwords, which might not be unique. In the event of password reuse, the attacker can generate a single log record to obtain the password and then use it to authenticate to all affected relying parties.

### 3 Logging for FIDO2

#### 3.1 Background

**FIDO2 protocol.** The FIDO2 protocol [36, 80] allows a client to authenticate using cryptographic keys stored on a device (e.g., a Yubikey hardware token or a Google passkey). To register with a relying party (e.g., `github.com`), the client generates an ECDSA keypair, stores the secret key, and sends the public

key to the relying party. When the client subsequently wants to authenticate to relying party `github.com`, Github’s server sends the client a random challenge. The client then signs the hash of the string `github.com` and the Github-chosen challenge using the secret key the client generated for `github.com` at registration. If the signature is valid, the Github server authorizes the client. Because the message signed by the client is bound to the name `github.com`, FIDO2 provides a strong defense against phishing attacks. The FIDO2 protocol supports passwordless, second-factor, and multi-factor authentication.

**Zero-knowledge arguments.** Informally, zero-knowledge arguments allow a prover to convince a verifier that a statement is true without revealing *why* the statement is true [43]. More precisely, we consider non-interactive zero-knowledge argument systems [13, 35] in the random-oracle model [10]. Both the prover and verifier hold the description of a computation  $C$  and a public input  $x$ . The prover’s goal is to produce a proof  $\pi$  that convinces the verifier that there exists a witness  $w$  that causes  $C(x, w) = 1$ , without revealing the witness  $w$  to the verifier. We require the standard notions of completeness, soundness, and zero knowledge [13, 43]. Throughout the paper, we will refer to this type of argument system as a “zero-knowledge proof.”

We use the ZKBoo protocol [54, 42, 20] for proving statements about computations expressed as Boolean circuits. Our system could also be instantiated with succinct non-interactive arguments of knowledge, which would decrease proof size and verification time, but at the cost of increasing proving time and requiring large parameters generated via a separate setup algorithm [12, 39, 45, 71].

**Threshold signatures.** A two-party threshold signature scheme [28, 29] is a set of protocols that allow two parties to jointly generate a single public key along with two shares of the corresponding secret key and then jointly sign messages using their secret key shares such that the signature verifies under the joint public key. Informally, no malicious party should be able to subvert the protocols to extract another party’s share of the secret key or forge a signature on a message other than the honest party’s message. We would ideally instantiate our system using BLS multisignatures [14]. Unfortunately, the FIDO2 standard limits the choice of signing algorithms to ECDSA and RSASSA [67]. For backwards-compatibility, we present a construction for two-party ECDSA signing with preprocessing tailored to our setting in Section 3.3.



### 3.2 Split-secret authentication

We now describe our split-secret authentication protocol for FIDO2 where the authentication secret is split between the larch client software and the log service. The key challenge is achieving log enforcement and log privacy simultaneously: every successful authentication should result in a valid log entry encrypting the identity of the relying party, but the log should not learn the identity of the relying party.

We use threshold signing to ensure that both the client and log participate in every successful authentication. A natural way to use threshold signing would be to have the client and log each generate a new threshold signing keypair at every registration. Unfortunately, if the log service used a different key share for each relying party, it would know which authentication requests correspond to the same relying party, violating Goal 2 (privacy against a malicious log). Instead, we have the log use the *same signing-key share for all relying parties*. The client still uses a different signing-key share per party, ensuring the public keys are unlinkable across relying parties. To authenticate to a relying party with identifier  $id$  and challenge  $chal$ , the client computes a digest  $dgst = \text{Hash}(id, chal)$  that hides  $id$ . The client and log then jointly sign  $dgst$ .

We also need to ensure that the log service obtains a correct record of every authentication. In particular, the log should only participate in threshold signing if it obtains a valid encryption  $ct$  of the relying-party identifier  $id$  [77].

To be valid, a ciphertext  $ct$  must (1) decrypt to  $id$  under the archive key  $k$  established for that client, and (2) be correctly related to the digest  $dgst$  that the log will sign (i.e.,  $\text{Dec}(k, ct) = id$  and  $dgst = \text{Hash}(id, chal)$ ). To allow the log service to check that the client is using the right archive key without learning the key, we use a commitment scheme. During enrollment, the client generates a commitment  $cm$  to the archive key  $k$  using random nonce  $r$  and sends  $cm$  to the log service. During authentication, the client uses a zero-knowledge proof to prove to the log that it knows a key  $k$ , randomness  $r$ , relying-party identifier  $id$ , and authentication challenge  $chal$  such that ciphertext  $ct$ , digest  $dgst$ , and commitment  $cm$  from enrollment meet the following conditions:

- (a)  $cm = \text{Commit}(k, r)$ ,
- (b)  $id = \text{Dec}(k, ct)$ , and
- (c)  $dgst = \text{Hash}(id, chal)$ .

The public inputs are the ciphertext  $ct$ , digest  $dgst$ , and commitment  $cm$  (known to the client and log); the witness is the archive key  $k$  (known only to the client), commitment opening  $r$ , relying-party identifier  $id$ , and challenge  $chal$ .

**Final protocol.** We now outline our final protocol.

*Enrollment.* During enrollment, the client samples a symmetric encryption key  $k$  as the archive key and commits to it with some random nonce  $r$ . The client sends the commitment  $cm$  to the log, and the log generates a signing-key share for the user. The log sends the client the public key corresponding

to its signing-key share to allow the client to derive future keypairs for relying parties.

*Registration.* At registration, the client generates a new signing-key share for that relying party. The client then aggregates the log's public key with its new signing-key share and sends the resulting public key to the relying party. No interaction with the log service is required.

*Authentication.* To authenticate to  $id$  with challenge  $chal$ , the client computes  $dgst \leftarrow \text{Hash}(id, chal)$  and  $ct \leftarrow \text{Enc}(k, id)$ . The client then generates a zero-knowledge proof  $\pi$  that it knows an archive key  $k$ , commitment nonce  $r$ , relying-party identifier  $id$ , and authentication challenge  $chal$  such that  $dgst$  and  $ct$  are correctly related relative to the commitment  $cm$  that the client generated at enrollment. The client sends  $dgst$ ,  $ct$ , and  $\pi$  to the log service. The log service checks the proof and, if it verifies, runs its part of the threshold signing protocol. The log service stores  $ct$  and returns its signature share to the client. The log service also stores the current time and client IP address with  $ct$ , allowing the user to obtain additional metadata by auditing. Finally, the client completes the threshold signature and sends it to the relying party.

*Auditing.* To audit the log, the client requests the list of ciphertexts and metadata from the log service and decrypts all of the relying-party identifiers.

### 3.3 Two-party ECDSA with preprocessing

Section 3.2 shows how to implement larch for any two-of-two threshold signing scheme that cryptographically hashes input messages. However, FIDO2 compatibility forces us to use ECDSA, which is more cumbersome than BLS to threshold. We present a concretely efficient protocol for ECDSA signing between the client and log.

There is a large body of prior work on multi-party ECDSA signing [31, 61, 22, 4, 23, 18, 48, 41, 40, 19]. However, existing protocols are orders of magnitude more costly than the one we present here [61, 41, 40, 18, 19]. The efficiency gain for us comes from the fact that we may assume that the client is *honest at enrollment time and only later compromised*. In contrast, standard schemes for two-party ECDSA signing must protect against the compromise of either party at any time. Prior protocols provide this stronger security property at a computational and communication cost. In our setting, we need only ensure that an honest client can run an enrollment procedure with the log service such that if the client is later compromised, the attacker cannot subvert the signing protocol.

We leverage the client to split signing into two phases:

1. During an *offline phase*, which takes place during enrollment, the client performs some preprocessing to produce a “presignature.” Security only holds if the client is honest during the offline phase.
2. During an *online phase*, which takes place during authentication, the client and log service use the presignature to

perform a lightweight, message-dependent computation to produce an ECDSA signature. Security holds if either the client or log service is compromised during the online phase.

Prior work also splits two-party signing into an offline and online phase. However, prior work performs this partitioning to reduce the online time at the expense of a more costly offline phase [22, 85, 23, 18]. (The offline phase in these schemes is expensive since the protocols do not assume that both parties are honest during the offline phase.) We split the signing scheme into an offline and online phase to take advantage of the fact that we may assume that the client is honest in the offline phase and so can reduce the total computation time this way.

An additional requirement in our setting is that the log should *not* learn the public key that the signature is generated under. Because the public key is specific to a relying party, hiding the public key is necessary for ensuring that the log cannot distinguish between relying parties. The signing algorithm can take as input a relying-party-specific key share from the client and a relying-party-independent key share from the log.

**Background: ECDSA.** For a group  $\mathbb{G}$  of prime order  $q$  with generator  $g$ , fixed in the ECDSA standard, an ECDSA secret key is of the form  $sk \in \mathbb{Z}_q$ , where  $\mathbb{Z}_q$  denotes the ring of integers modulo  $q$ . The corresponding ECDSA public key is  $pk = g^{sk} \in \mathbb{G}$ . ECDSA uses a hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \mathbb{Z}_q$  and a “conversion” function  $f: \mathbb{G} \rightarrow \mathbb{Z}_q$ . To generate an ECDSA signature on a message  $m \in \{0, 1\}^*$  with secret key  $sk \in \mathbb{Z}_q$ , the signer samples a signing nonce  $r \xleftarrow{\mathbb{R}} \mathbb{Z}_q$  and computes

$$r^{-1} \cdot (\text{Hash}(m) + f(g^r) \cdot sk) \in \mathbb{Z}_q.$$

**Our construction.** We now describe our construction for a two-party ECDSA signing protocol with presignatures. (See the full version [26] for technical details.) To generate the log keypair, the log samples  $x \xleftarrow{\mathbb{R}} \mathbb{Z}_q$ , sets its secret key to  $x \in \mathbb{Z}_q$ , and sets its public key to  $X = g^x \in \mathbb{G}$ . Then to generate a keypair from the log public key, the client samples  $y \xleftarrow{\mathbb{R}} \mathbb{Z}_q$  and sets the relying-party-specific public key to  $pk = X \cdot g^y \in \mathbb{G}$ . For each public key of the form  $g^{x+y} \in \mathbb{G}$ , the log has one share  $x \in \mathbb{Z}_q$  of the secret key that is the same for all public keys and the client has the other share  $y \in \mathbb{Z}_q$  of the secret key that is different for each public key.

We split the signature-generation process into two parts:

1. *Offline phase:* a message-independent, key-independent “presignature” algorithm that the client runs, and
2. *Online phase:* a message-dependent, key-dependent signing protocol that the log and client run jointly.

To generate the presignature in the offline phase, the client samples a signing nonce  $r \xleftarrow{\mathbb{R}} \mathbb{Z}_q$ , computes  $R \leftarrow g^r \in \mathbb{G}$ , and splits  $r^{-1}$  into additive secret shares:  $r^{-1} = r_0 + r_1 \in \mathbb{Z}_q$ . The log’s portion of the presignature is  $(f(R), r_0) \in \mathbb{Z}_q^2$ , and the client’s portion is  $(f(R), r_1) \in \mathbb{Z}_q^2$ . Then, to produce a signature on a message in the online phase, the client and

log simply perform a single secure multiplication to compute

$$r^{-1} \cdot (\text{Hash}(m) + f(R) \cdot sk) \in \mathbb{Z}_q$$

where  $r^{-1} \in \mathbb{Z}_q$  (signing nonce) and  $sk \in \mathbb{Z}_q$  (signing key) are secret-shared between the client and log.

To perform this multiplication over secret-shared values, we use Beaver triples [9]. A Beaver triple is a set of one-time-use shares of values that the log and client can use to efficiently perform a two-party multiplication on secret-shared values. Traditionally, generating Beaver triples is one of the expensive portions of multiparty computation protocols (e.g., in prior work on threshold ECDSA [22]). In our setting, the client at enrollment time can generate a Beaver triple as part of the presignature. Note that the client and log can use each signing nonce and Beaver triple exactly once. That is, the client and log must use a fresh presignature to generate each signature.

**Malicious security.** By deviating from the protocol, neither the client nor the log should be able to learn secret information (i.e., the other party’s share of the secret key or signing nonce) or produce a signature for any message apart from the one that the protocol fixes. We describe how to accomplish this using traditional tools for malicious security (e.g. information-theoretic MACs [24]) in the full version [26].

**Formalizing and proving security.** We define and prove security in the full version [26].

**Implications for system design.** Our preprocessing approach increases the client’s work at enrollment: the client generates some number of presignatures (e.g., 10K) and sends the log’s presignature shares to the log. To reduce storage burden on the log, the client can store encryptions of the log’s presignature shares.

When the client is close to running out of presignatures, it can authenticate with the log, generate more presignatures, and send the log’s presignature shares to the log service. If the log service does not receive an objection after some period of time, it will start using the new presignatures. An honest client periodically checks the log to see whether any unexpected presignatures (created by an attacker) appear in its log. If the client learns that a new batch of presignatures was generated that the client did not authorize, the client authenticates to the log service and objects.

If the client runs out of presignatures and the log service rejects the client’s presignatures, the client and the log can temporarily use a more expensive signing protocol that does not require presignatures [41, 61, 31, 85]. The client could run out of presignatures and be forced to use the slow multisignature protocol in the following cases:

1. The attacker compromised the user’s credentials with the log service, allowing the attacker to object to the new presignatures. In this case, the attacker could change the user’s credentials and permanently lock the user out of her account.

2. The honest client was close to running out of presignatures, generated new presignatures, and then ran out of presignatures while waiting for a possible objection. This scenario only occurs when the honest client makes an unexpectedly large number of authentications in a short period of time. The client only needs to pay the cost of the slow multisignature protocol for a short period of time.

An attacker that has compromised the log service can also deny service, as we discussed in Section 2.4.

## 4 Logging for time-based one-time passwords

We now show how larch can support time-based one-time passwords (TOTP).

### 4.1 Background: TOTP

TOTP is a popular form of second-factor authentication that authenticator apps (Authy, Google Authenticator, and others [68]) implement. When a client registers for TOTP with a relying party, the relying party sends the client a secret cryptographic key. Then, to authenticate, the client and the relying party both compute a MAC on the current time using the secret key from registration. The client sends the resulting MAC tag to the server. If the client’s submitted tag matches the one that the server computes, the relying party authorizes the client. TOTP uses a hash-based MAC (HMAC).

### 4.2 Split-secret authentication for TOTP

At a high level, in our split-secret authentication protocol for TOTP, both the client and log service have as private input additive secret shares of the TOTP secret key. At the conclusion of the split-secret authentication, the client holds a TOTP code and the log service holds a ciphertext. We now give the details of our protocol.

*Enrollment.* At enrollment, just as with FIDO2, the client generates and stores a long-term symmetric-encryption archive key  $k$  and random nonce  $r$ . Then, the client sends the commitment  $cm = \text{Commit}(k, r)$  to the log service.

*Registration.* To register a client, a relying party generates and sends the client a secret MAC key  $k_{id}$  for TOTP. The client samples a random identifier  $id$  for the relying party and then splits the TOTP secret key  $k_{id}$  into additive secret shares  $klog_{id}$  and  $kclient_{id}$ . The client sends  $(id, klog_{id})$  to the log service and locally stores  $(id, kclient_{id})$  alongside a name identifying the relying party (e.g., `user@amazon.com`).

*Authentication.* In order to authenticate to the relying party  $id$  at time  $t$ , the client needs to compute  $\text{HMAC}(k_{id}, t)$  with the help of the log service. Let  $n$  be the number of relying parties with which the client has registered. To authenticate, the client and log service run a secure two-party computation where:

- The client’s input is its long-term symmetric archive key  $k$  and commitment opening  $r$  from enrollment, the relying-party identifier  $id$ , and the client’s share of the TOTP key  $kclient_{id}$ .
- The log service’s input is the commitment  $cm$  from enrollment, the list of relying-party identifiers that the client has registered with  $(id_1, \dots, id_n)$ , and the log service’s TOTP key shares  $(klog_{id_1}, \dots, klog_{id_n})$ —one per relying party.
- The client outputs the TOTP code  $\text{HMAC}(k_{id}, t)$ .
- The log outputs an encrypted log record: an encryption of the relying-party identifier  $id$  under the archive key  $k$ .

We execute this two-party computation using an off-the-shelf garbled-circuit-based multiparty computation protocol. Garbled circuits allow two parties to jointly execute any Boolean circuit on private inputs, where neither party learns information about the other’s input beyond what they can infer from the circuit’s output [87]. We use the protocol from Wang et al. [84], which provides malicious security, meaning that the protocol remains secure even if one corrupted party deviates arbitrarily from the protocol. As long as either the client or the log service is honest, the log service does not learn any information about the client’s authentication secrets, and the client learn no information about the TOTP secret, apart from the single TOTP code that the protocol outputs. Because we use an off-the-shelf garbled-circuit protocol, the communication overhead is much higher than in the special-purpose protocols we design for FIDO2 and passwords (Section 8). TOTP is challenging to design a special-purpose protocol for because the authentication credential must be generated via the SHA hash function which, unlike the authentication credentials for FIDO2 and passwords, does not have structure we can exploit. Clients can ask the log service to delete registrations for unused accounts to speed up the two-party computation.

*Auditing.* To audit the log, the client simply requests the list of ciphertexts from the log service. The client decrypts each ciphertext with its archive key  $k$  and then, using its mapping of  $id$  values to relying party names, outputs the resulting list of relying party names.

## 5 Logging for passwords

We now describe how larch can support passwords.

### 5.1 Protocol overview

We construct a split-secret authentication protocol that takes place between the client and the log service. In particular, we show how the client can compute the password to authenticate to a relying party in such a way that (a) the log service does not learn the relying party’s identity and (b) the client’s authentication attempt is logged. At the start of the authentication protocol run:

- the client holds a secret key, the log service’s public key, and the identity  $\text{id}^*$  of the relying party it wants to authenticate to, and
- the log service holds its own secret key, the client’s public key, and a list of relying-party identities  $(\text{id}_1, \dots, \text{id}_n)$  at which the client has registered.

At the end of the authentication protocol run:

- the client holds a password derived as a pseudorandom function of the client’s secret, the log’s secret, and the relying party identity  $\text{id}^*$ , and
- the log service holds a ciphertext encrypting the relying party’s identity  $\text{id}^*$  under the client’s public key.

**Limitations inherent to passwords.** As we discussed in Section 2.4, larch for passwords does not protect against credential breaches, but does defend against device compromise.

## 5.2 Split-secret authentication for passwords

The larch scheme for password-based authentication uses a cyclic group  $\mathbb{G}$  of prime order  $q$  with a fixed generator  $g \in \mathbb{G}$ . Our implementation uses the NIST P-256 elliptic-curve group.

When using password-based authentication in larch, the client and log service after registration each hold a secret share of the password for each relying party. In particular, the password for a relying party with identity  $\text{id} \in \{0, 1\}^*$  is the string  $\text{pw}_{\text{id}} = k_{\text{id}} \cdot \text{Hash}(\text{id})^k \in \mathbb{G}$ , where:

- $k_{\text{id}} \in \mathbb{Z}_q$  is a per-relying-party secret share held by the client,
- $\text{Hash}: \{0, 1\}^* \rightarrow \mathbb{G}$  is a hash function, and
- $k \in \mathbb{Z}_q$  is a per-client secret key held by the log service.

Thus, computing  $\text{pw}_{\text{id}}$  requires both the client’s per-site key  $k_{\text{id}}$  and the log’s secret key  $k$ .

The technical challenge is to construct a protocol that allows the client to compute the password  $\text{pw}_{\text{id}}$  while (a) hiding  $\text{id}$  from the log service and (b) ensuring that the log service completes the interaction holding an encryption of  $\text{id}$  under the client’s public key.

**Protocol.** We describe the protocol steps:

*Enrollment.* The client samples an ElGamal secret key  $x \in \mathbb{Z}_q$  as the archive key and sends the corresponding public key  $X = g^x \in \mathbb{G}$  to the log service. The log service samples a Diffie-Hellman secret key  $k \in \mathbb{Z}_q$  and sends its public key  $K = g^k \in \mathbb{G}$  to the client.

*Registration.* The client samples a per-relying-party random identifier  $\text{id} \leftarrow \{0, 1\}^{128}$ , saves  $\text{id}$  locally alongside the name of the relying party (e.g., `user@amazon.com`), and sends  $\text{id}$  to the log service. The log service saves the string  $\text{Hash}(\text{id})$  and replies with  $\text{Hash}(\text{id})^k \in \mathbb{G}$ . To generate a new strong password  $\text{pw}_{\text{id}}$  (the recommended use), the client samples and saves a random key share  $k_{\text{id}} \leftarrow \mathbb{G}$  and sets  $\text{pw}_{\text{id}} \leftarrow k_{\text{id}} \cdot \text{Hash}(\text{id})^k \in \mathbb{G}$ . To import a legacy password  $\text{pw}_{\text{id}}$  (less secure), the client

computes and stores  $k_{\text{id}} \leftarrow \text{pw}_{\text{id}} \cdot (\text{Hash}(\text{id})^k)^{-1} \in \mathbb{G}$ . The client then deletes  $\text{Hash}(\text{id})^k$  and  $\text{pw}_{\text{id}}$ . Note that the log server can discard  $\text{id}$ , which it only uses to avoid providing  $h^k$  for arbitrary  $h$ . When the client samples  $\text{id}$  and  $k_{\text{id}}$  randomly in the recommended usage, the password  $\text{pw}_{\text{id}}$  for each relying party is random and distinct.

*Authentication.* During authentication, the client must recompute the password  $\text{pw}_{\text{id}}$ . To do so, the client first sends the log service an encryption of  $\text{Hash}(\text{id})$  under the public ElGamal archive key  $g^x$ : the client samples  $r \leftarrow \mathbb{Z}_q^*$  and computes the ciphertext  $(c_1, c_2) = (g^r, \text{Hash}(\text{id}) \cdot g^{xr}) \in \mathbb{G}^2$ . In addition, the client sends a zero-knowledge proof to the log service attesting to the fact that  $(c_1, c_2)$  is an encryption under the client’s public key  $X$  of  $\text{Hash}(\text{id})$  for  $\text{id} \in \{\text{id}_1, \text{id}_2, \dots, \text{id}_n\}$ —the set of relying-party identifiers that the client sent to the log service during each of its registrations so far. The client executes this proof using the technique from Groth and Kohlweiss [46]. The proof size is  $O(\log n)$  and the prover and verifier time are both  $O(n)$ . (See the full version [26] for implementation details.)

The log service saves the ciphertext as a log entry, checks the zero-knowledge proof, and returns the value  $h = c_2^k = \text{Hash}(\text{id})^k \cdot g^{xrk} \in \mathbb{G}$  to the client. The client can then compute

$$\text{pw}_{\text{id}} = k_{\text{id}} \cdot h \cdot K^{-xr} = k_{\text{id}} \cdot \text{Hash}(\text{id})^k \in \mathbb{G}.$$

Crucially, the client deletes  $\text{pw}_{\text{id}}$  after authentication to ensure that future authentications must again interact with the log service.

*Auditing.* To audit the log, the client downloads the ElGamal ciphertexts and can decrypt each ciphertext to recover a list of hashed identities:  $(\text{Hash}(\text{id}_1), \text{Hash}(\text{id}_2), \dots)$ . The client uses its stored mapping of ids to relying-party identifiers to recover the plaintext names of the relying parties in the log.

## 6 Protecting against log misbehavior

The larch log service must participate in each of the user’s authentication attempts. If the log service goes offline, the user will not be able to authenticate to any of her larch-enabled relying parties. In a real-world deployment, the log service could consist of multiple servers replicated using standard state-machine replication techniques to tolerate benign failures [58, 70]. However, users might also worry about intentional denial-of-service attacks on the part of the log.

To defend against availability attacks, a user can split trust across multiple logs. At enrollment time, the user can enroll with  $n$  logs. Then at registration, the user can set a threshold  $t$  of logs that must participate in authentication. Thus, the user can authenticate to her accounts so long as  $t$  logs are online, and she can audit activity so long as  $n - t + 1$  logs are available. We need  $n - t + 1$  logs to be available for auditing in order to guarantee that at least one of the  $t$  logs that participated in authentication is online. To ensure that colluding logs cannot authenticate on behalf of a client, the user’s client



can run  $n + 1$  logical parties, and  $n + t + 1$  parties can generate an authentication credential. In the setting with multiple log services, we need to adapt our two-party protocols to threshold multi-party protocols. Although we present our techniques for two parties (the client and a single log), our techniques generalize to multiple parties in a straightforward way.

For FIDO2 and passwords, the client now sends a zero-knowledge proof to each of the  $n$  logs. In the password case, the client can then retrieve  $(t, n)$  Shamir shares of the password [75], and in the FIDO2 case, the client can run any existing multi-party threshold signing protocol that does not take the public key as input [76, 22]. For TOTP, the client and the  $n$  logs can execute the same circuit using any malicious-secure threshold multi-party computation protocol [11].

Note that for relying parties that support FIDO2, users can optionally register a backup hardware FIDO2 device to allow them to bypass the log. In this case, the user can authenticate either via larch or via her backup FIDO2 key. While registering a backup hardware device protects against availability attacks, if an attacker obtains this hardware device, they can authenticate as the user without interacting with the log.

## 7 Implementation

We implemented larch for FIDO2, TOTP, and passwords with a single log service. We use C/C++ with gRPC and OpenSSL with the P256 curve (required by the FIDO2 standard). We wrote approximately 5,700 lines of C/C++ and 50 lines of Javascript (excluding tests and benchmarks). Our implementation is available at <https://github.com/edauterman/larch>.

For our FIDO2 implementation, we implemented a ZK-Boo [42] library for arbitrary Boolean circuits. Our ZKBoo implementation (with optimizations from ZKB++ [20]) uses emp-toolkit to support arbitrary Boolean circuits in Bristol Fashion [83]. To support the parallel repetitions required for soundness error  $< 2^{-80}$ , we use SIMD instructions with a bitwidth of 32 and run 5 threads in parallel. For the proof circuit, we use AES in counter mode for encryption and SHA-256 for commitments (SHA-256 is necessary for backwards compatibility with FIDO2). We built a log service and client that invoke the ZKBoo library, as well as a Chrome browser extension that interfaces with our client application and is compatible with existing FIDO2 relying parties. We built our browser extension on top of an existing extension [56].

Our TOTP implementation uses a maliciously secure garbled-circuit construction [84] implemented in emp-toolkit [83]. We generated our circuit using the CBMC-GC compiler [37] with ChaCha20 for encryption and SHA-256 for commitments.

For our passwords implementation, we implemented Groth and Kohlweiss’s proof system [46].

Our implementation uses a single log server for the log service, does not encrypt communication between the client and the log service, and does not require the client to

authenticate with the log service. A real-world deployment would use multiple servers for replication, use TLS between the client and the log service, and authenticate the client before performing any operations.

**Optimizations.** We use pseudorandom generators (PRGs) to compress presignatures: the log stores 6 elements in  $\mathbb{Z}_q$  and the client stores 1 element. Also, instead of running an authenticated encryption scheme (e.g. AES-GCM) inside the circuit for FIDO2 or TOTP, we run an encryption scheme without authentication (e.g. AES in counter mode) inside the circuit and then sign the ciphertext (client has the signing key, log has the verification key). The log can check the integrity of the ciphertext by verifying the signature, which is must faster than checking in a zero-knowledge proof or computing the ciphertext tag jointly in a two-party computation.

## 8 Evaluation

In this section, we evaluate the cost of larch to end users and the cost of running a larch log service.

**Experiment setup.** We run our benchmarks on Amazon AWS EC2 instances. Unless otherwise specified, we run the log service on a c5.4xlarge instance with 8 cores (2 hyperthreads per core) and 32GB of memory and, for latency benchmarks, the client on a c5.2xlarge instance with 4 cores and 16GB of memory, comparable to a commodity laptop. We configure the network connection between the client and log service to have a 20ms RTT and a bandwidth of 100 Mbps.

### 8.1 End-user cost

We show larch authentication latency and communication costs for FIDO2, TOTP, and passwords.

#### 8.1.1 FIDO2

**Latency.** The client for our FIDO2 scheme can complete authentication in 303ms with a single CPU core, or 117ms when using eight cores (Figure 3). Loading a webpage often takes a few seconds because of network latency, so the client cost of larch authentication is minor by comparison. The client’s running time during authentication is independent of the number of relying parties. The heaviest part of the client’s computation is proving to the log service that its encrypted log entry is well formed.

At enrollment, the client must generate many “presignatures,” which it later uses to run our authentication protocol with the log. Generating 10,000 presignatures for 10,000 future FIDO2 authentications takes 885ms. When the client runs out of presignatures, it generates new presignatures it can use after a waiting period (see Section 3.3).

**Communication.** During enrollment, the client must send the log 1.8MiB worth of presignatures. Thereafter, each authenti-

ation attempt requires 1.73MiB worth of communication: the bulk of this consists of the client’s zero-knowledge proof of correctness, and 352B of it comes from the signature protocol. By using a different zero-knowledge proof system, we could reduce communication cost at the expense of increasing client computation cost.

**Comparison to existing two-party ECDSA.** For comparison, a state-of-the-art two-party ECDSA protocol [85] that does not require presignatures from the client and uses Paillier requires 226ms of computation at signing time (the authors’ measurements exclude network latency, which we estimate would add 80ms) and 6.3KiB of per-signature communication. In contrast, our signing protocol only requires 0.5KiB per-signature communication (including the log presignature and the signing messages) and takes 61ms time at signing, almost all of which is due to network latency and can be run in parallel with proving and verifying as the computational overhead is minimal (1ms).

### 8.1.2 TOTP

**Latency.** In Figure 3 (right), we show how TOTP authentication latency increases with the number of relying parties the user registered with. Because we implement TOTP authentication using garbled circuits [84], we can split authentication into two phases: an “offline”, input-independent phase and an “online”, input-dependent phase (the log service and client communicate in both phases). Both phases are performed once per authentication. However, the offline phase can be performed in advance of when the user needs to authenticate to their account, and so it does not affect the latency that the user experiences. For 20 relying parties, the online time is 91ms and the offline time is 1.23s. For 100 relying parties, the online time is 120ms and the offline time is 1.39s.

**Communication.** Communication costs for our TOTP authentication scheme are large: for 20 relying parties, the total communication cost is 65MiB, and for 100 relying parties, the total communication is 93MiB. The online communication costs are much smaller: for 20 relying parties, the online communication is 202KiB and for 100 relying parties, the online communication is 908KiB. We envision clients running the offline phase in the background while they have good connectivity. While these communication costs are much higher than those associated with FIDO2 or passwords, we expect users to authenticate with TOTP less frequently because TOTP is only used for second-factor authentication.

### 8.1.3 Passwords

**Latency.** In Figure 3 (center), we show how password authentication latency increases with the number of registered relying parties. With 16 relying parties, authentication takes 28ms, and with 512 relying parties, it takes 245ms: the authentication time grows linearly with the number of relying parties. The proof system we use requires padding the number of relying parties to

the nearest power of two, meaning that registering at additional relying parties does not affect the latency or communication until the number of relying parties reaches the next power of two.

**Communication.** In Figure 5, we show how communication increases logarithmically with the number of relying parties. This behavior is due to the fact that proof size is logarithmic in the number of relying parties. With 16 relying parties, the communication is 1.47KiB, and with 512 relying parties, it is 4.14KiB.

## 8.2 Cost to deploy a larch service

If successful, larch can become much simpler and more efficient with a little support from future FIDO specifications (see Section 9). Nonetheless, we show larch is already practical by analyzing the cost of deploying a larch service today (Table 6). We expect a larch log service to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of TOTP authentications. This is because the majority of relying parties only support passwords, and relying parties typically require second-factor authentication only from time to time.

Throughout this section, we consider password-based authentication with 128 relying parties (based on the fact that the average user has roughly 100 passwords [73]) and TOTP-based authentication with 20 relying parties (based on the fact that Yubikey’s maximum number of TOTP registrations is 32 [2]). The authentication overhead of FIDO2 in larch is independent of the number of relying parties the user has registered with.

**Storage.** For each of the three protocols, the log service must store authentication records (timestamp, ciphertext, and signature). FIDO2 and TOTP have 88B authentication records, and passwords have 138B records (due to the size of ElGamal ciphertexts). The FIDO2 protocol additionally requires the client to generate presignatures for the log, each of which is 192B. For 10K presignatures, the log service must store 1.83MiB. In Figure 4 (left), we show how per-client log storage actually decreases as presignatures are consumed and replaced by authentication records. To minimize storage costs, the log service can encrypt its presignatures and store them at the client. The log service then simply needs to keep a counter to prevent presignature re-use.

**Throughput.** In Table 6, we show the number of auths/s a single log service core can support assuming 128 passwords and 20 TOTP accounts. We achieve the highest throughput for passwords (47.62 auths/cores/s), which are the most common authentication mechanism. For FIDO2, which can be used as either a first or second authentication factor and is supported by fewer relying parties than passwords, we achieve 6.18 auths/core/s. Finally, for TOTP, which is only used as a second factor, we achieve 0.73 auths/core/s.

Our FIDO2 protocol can be instantiated with any NIZK proof system to achieve a different tradeoff between authentication latency and log service throughput. For example, we instan-

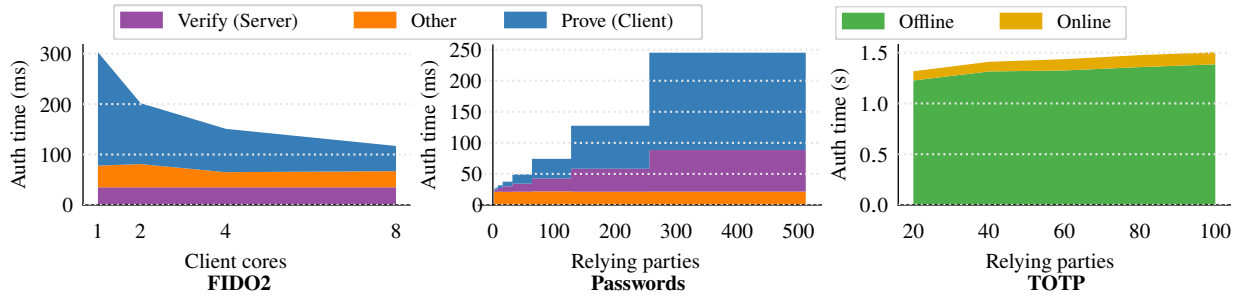


Figure 3: On the left, larch FIDO2 latency decreases as the number of client cores increases (latency is independent of the number of relying parties). In the center, larch password latency grows with the number of relying parties, with the majority of the time spent on client proof generation. On the right, larch TOTP latency grows with the number of relying parties, with the majority of the time spent in an input-independent “offline” phase as opposed to the input-dependent “online” phase (both phases require network communication).

tiate our system with ZKBoo, but could also use Groth16 [45] to reduce communication and verifier time (increasing log throughput). We measure the performance of Groth16 on our larch FIDO2 circuit on the BN-128 curve using ZoKrates [91] with libsnark [57] with a single core (we only measure the overhead of SHA-256, which dominates circuit cost, to provide a performance lower bound). While the verifier time is much lower (8ms) and the proof is much smaller (4.26KiB), (1) the trusted setup requires the client to store 19.86MiB and the log service to store 9.2MiB per client, and (2) the proving time is 4.07s, meaning that authentication latency is much higher.

**Cost.** We now quantify the cost of running a larch log service. The cost of one core on a c5 instance is \$0.0425-\$0.085/hour depending on instance size [1]. Data transfer to AWS instances is free, and data transfer from AWS instances costs \$0.05-\$0.09/GB depending on the amount of data transferred per month [1]. In Table 6, we show the cost of supporting 10M authentications for each authentication method with larch.

Supporting 10M authentications requires 450 log core hours for FIDO2, 3,832 log core hours for TOTP, and 59 log core hours for passwords. Compute for 10M authentications costs \$19.13-\$38.25 for FIDO2, \$162.86-\$325.72 for TOTP, and \$2.51-\$5.02 for passwords. Communication for 10M authentications costs \$0.10-\$0.19 for FIDO2, \$17,923-\$32,262 for TOTP, and \$0.015-\$0.027 for passwords. The high cost for TOTP is due to the large amount of communication required at authentication: the log service must send the client 36.8MiB for every authentication. In both the FIDO2 and password protocols, the vast majority of the communication overhead is due to the proof sent from the client to the log service, which incurs no monetary cost. We show how cost increases with the number of authentications for each of the the authentication methods in Figure 4 (right).

TOTP is substantially more expensive than FIDO2 or passwords. However, we expect a relatively small fraction of authentication requests to be for TOTP.

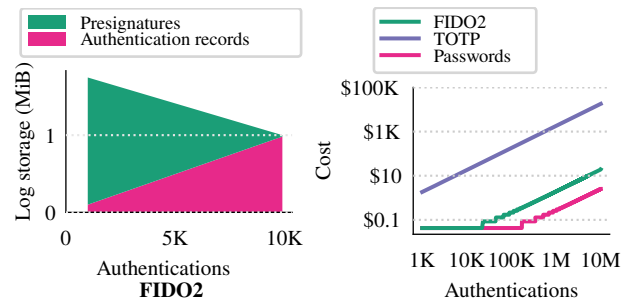


Figure 4: On the left, per-client storage overhead at the log decreases as presignatures are replaced with authentication records (client enrolls with 10K presignatures). On the right, minimum cost of supporting more authentications with passwords, (128 relying parties), FIDO2, and TOTP (20 relying parties). Both axes use a logarithmic scale.

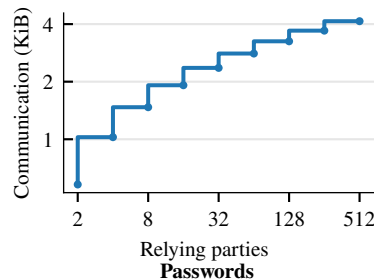


Figure 5: Communication for larch with passwords increases logarithmically with the number of relying parties (both axes use a logarithmic scale).

## 9 Discussion

**Deployment strategy.** Because larch supports passwords, TOTP, and FIDO2, people can use it with the vast majority of web services. In addition, larch offers users many of the benefits of FIDO2 without a dedicated hardware security token, particularly FIDO2’s protection against phishing. The flexibility for users to choose log services can foster an ecosystem of new security products, such as log services that request login confirmation via a mobile phone app, apps that monitor the log to notify users of anomalous behavior, or enterprise security products that monitor access to arbitrary

	FIDO2	TOTP	Password
Online auth time	150 ms	91 ms	74 ms
Total auth time	150 ms	1.32 s	74 ms
Online auth comm.	1.73 MiB	201 KiB	3.25 KiB
Total auth comm.	1.73 MiB	65 MiB	3.25 KiB
Auth record	88 B	88 B	138 B
Log presignature	192 B	∅	∅
Log auths/core/s	6.18	0.73	47.62
10M auths min cost	\$19.19	\$18,086	\$2.48
10M auths max cost	\$38.37	\$32,588	\$4.96

Table 6: Costs for larch with FIDO2, TOTP (20 relying parties), and passwords (128 relying parties). We take the cost of one core on a c5 instance to be \$0.0425-\$0.085/hour (depending on instance size) and data transfer out of AWS to cost \$0.05-\$0.09/GB (depending on amount of data transferred) [1]. For comparison, the Argon2 password hash function should take 0.5s using 2 cores.

third-party services that a company could contract with.

**FIDO improvements.** Larch can benefit from enhancements we hope to see considered for future versions of the FIDO specification. One simple improvement would be to support BLS signatures, which are easier to threshold and so eliminate larch’s need for presignatures [14].

Future versions of FIDO could also directly support secure client-side logging by allowing the relying party to compute the encrypted log record itself. The relying party could then ensure that the log service receives the correct encrypted log record by checking for the log record in the signing payload. Specifically, the signature payload could have the form:

Hash(log-record-ciphertext, Hash(remaining-FIDO-data)) .

The log server can then take the outer hash preimage as input without needing to verify anything else about the log record.

We want to allow the relying party to generate the encrypted log record without making it possible to link users across relying parties. Instead of giving the relying party the user’s public key directly at registration, which would link a user’s identity across relying parties, we instead give the relying party a key-private, re-randomizable encryption of the relying party’s identifier (we can achieve this using ElGamal encryption). At authentication, the relying party can re-randomize the ciphertext to generate the encrypted log record.

We also hope that future FIDO revisions standardize and promote authentication metadata as part of the challenge and hypothetical log record field. For users with multiple accounts at one relying party, it would be useful to include account names as well as relying party names in signed payloads. It would furthermore improve security to allow distinct types of authentication log records for different security-sensitive operations such as authorizing payments and changing or

removing 2FA on an account. An app monitoring a user’s log can then immediately notify the user of such operations.

**Multiple devices.** Clients need to authenticate to their accounts across multiple devices, which requires synchronizing a small amount of dynamic, secret state across devices. Cross-device state could be stored encrypted at the log, or could be disseminated through existing profile synchronization mechanisms in browsers. There is a danger of the synchronization mechanism maliciously convincing two devices to use the same presignature. Therefore, presignatures should be partitioned between devices in advance, and devices should employ techniques such as fork consistency [65] to detect and deter any rollback attacks. Existing tools can help a user recover if she loses all of her devices [25, 55, 81, 62].

**Enforcing client-specific policies.** We can extend larch in a straightforward way to allow the log to enforce more complex policies on authentications. The client could submit a policy at enrollment time, and the log service could then enforce this policy for subsequent authentications. If the policy decision is based on public information, the log service can apply the policy directly (e.g., rate-limiting, sending push notifications to a client’s mobile device). Other policies could be based on private information. For example, if we used larch for cryptocurrency wallets, the log could enforce a policy such as “deny transactions sending more than \$10K to addresses that are not on the allowlist.” For policies based on private information, the client could send the log service a commitment to the policy at enrollment, and the log service could then enforce the policy by running a two-party computation or checking a zero-knowledge proof.

**Revocation and migration.** If a client loses her device or wants to migrate her authentication secrets from an old device to a new device, she needs a way to easily and remotely invalidate the secrets on the old device. Larch allows her to do this easily. To migrate credentials to a new device, the client and log simply re-share the authentication secrets. To invalidate the secrets on the old device, the client asks the log to delete the old secret shares (client must authenticate with the log first).

**Account recovery.** In the event that a client loses all of her devices, she needs some way to recover her larch account. To ensure that she can later recover her account, the client can encrypt her larch client state under a key derived from her password and store the ciphertext with the larch service. The security of the backup is only as good as the security of the client’s password. Alternatively, the client could choose a random key to encrypt her client state and then back up this key using her password and secure hardware in order to defend against password-guessing attacks [25].

**Limitations.** If an attacker compromises the client’s account with the log, the attacker can access the client’s entire authentication history. To mitigate this damage, the log could delete old authentication records (e.g., records older than one week) or re-encrypt them under a key that the user keeps offline.



## 10 Related work

**Privacy-preserving single sign-on.** Like larch, existing privacy-preserving single sign-on systems hide the relying party from the identity provider. Unlike larch, these systems do not protect users' accounts from a malicious attacker that compromises the identity provider, and they do not privately log the identity of the relying party. BrowserID [33] (implemented in Mozilla Persona and Firefox Accounts) and SPRESSO [34] are single sign-on services that ensure that the identity provider does not learn the identity of the relying parties. However, neither prevents colluding relying parties from linking a user's accounts across relying parties. EL PASSO [90], UnlimitID [53], UPRESSO [50], PseudoID [30] and Hammann et al. [51] show how to build single sign-on services that protect clients from curious identity providers while ensuring that relying parties cannot link users' accounts.

Separately, Privacy Pass allows a user to obtain anonymous tokens for completing CAPTCHAs, which she can then spend at different relying parties without allowing them to link her across sites [27]. Like larch, Privacy Pass does not link users across accounts, but unlike larch, Privacy Pass does not provide a mechanism for logging authentications.

**Threshold signing.** Our two-party ECDSA with preprocessing protocol builds on prior work on threshold ECDSA. MacKenzie and Reiter proposed the first threshold ECDSA protocol for a dishonest majority specific to the two-party setting [64]. Genarro et al. [41] and Lindell [61] subsequently improved on this protocol. Doerner et al. show how to achieve two-party threshold ECDSA without additional assumptions [31]. Another line of work supports threshold ECDSA using generic multi-party computation over finite fields [76, 22]. A number of works show how to split ECDSA signature generation into online and offline phases [23, 18, 48, 47, 38, 19, 85, 4]; in many, the offline phase is signing-key-specific, allowing for a non-interactive online signing phase, whereas we need an offline phase that is signing-key-independent. Abram et al. show how to reduce the bandwidth of the offline phase via pseudorandom correlation generators [4]. Aumasson et al. provide a survey of prior work on threshold ECDSA [8]. Arora et al. show how to split trust across a group of FIDO authenticators to enable account recovery using a new group signature scheme [6].

**Proving properties of encrypted data.** Larch's split-secret authentication protocol for FIDO2 and passwords relies on proving properties of encrypted data, which is also explored in prior work. Verifiable encryption was first proposed by Stadler [77], and Camenisch and Damgard introduced it as a well-defined primitive [15]. Subsequent work has designed verifiable encryption schemes for limited classes of relations (e.g. discrete logarithms) [16, 7, 86, 63, 69]. Takahashi and Zaverucha introduced a generic compiler for MPC-in-the-head-based verifiable encryption [78]. Lee et al. [60] contribute a SNARK-based verifiable encryption scheme that decouples the encryption function from the circuit

by using a commit-and-prove SNARK [17]. This approach does not work for us for FIDO2 authentication because the ciphertext must be connected to a SHA-256 digest.

Grubbs et al. introduce zero-knowledge middleboxes, which enforce properties on encrypted data using SNARKs [49]. Wang et al. show how to build blind certificate authorities, enabling a certificate authority to validate an identity and generate a certificate for it without learning the identity [82]. DECO allows users to prove that a piece of data accessed via TLS came from a particular website and, optionally, prove statements about the data in zero-knowledge [89].

**Transparency logs.** Like larch, transparency logs detect attacks rather than prevent them, and they achieve this by maintaining a log recording sensitive actions [66, 44, 52, 21, 59, 5, 25]. However, transparency logs traditionally maintain public, global state. For example, the certificate transparency log records what certificates were issued and by whom in order to track when certificates were issued incorrectly [59]. In contrast, the larch log service maintains encrypted, per-user state about individual users' authentication history.

## 11 Conclusion

Larch is an authentication manager that logs every successful authentication to any of a user's accounts on a third-party log service. It guarantees log integrity without trusting clients. It furthermore guarantees account security and privacy without trusting the log service. Larch works with any existing service supporting FIDO2, TOTP, or password-based login. Our evaluation shows the implementation is practical and cost-effective.

**Acknowledgements.** We thank the anonymous reviewers and our shepherd Ittay Eyal for their feedback. We also thank Raluca Ada Popa for her support, as well as students in the Sky security group for giving feedback that improved the presentation of this paper. The RISELab and Sky Lab are supported by NSF CISE Expeditions Award CCF-1730628 and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Emma Dauterman was supported by an NSF Graduate Research Fellowship and a Microsoft Ada Lovelace Research Fellowship. This work was funded in part by the Stanford Future of Digital Currency Initiative as well as gifts from Capital One, Facebook, Google, Mozilla, Seagate, and MIT's FinTech@CSAIL Initiative. We also received support under NSF Award CNS-2054869.

## References

- [1] Amazon EC2 On-Demand Pricing. <https://aws.amazon.com/ec2/pricing/on-demand/>, accessed December 7, 2022.

- [2] How many accounts can I register my YubiKey with?, 2020. <https://support.yubico.com/hc/en-us/articles/360013790319-How-many-accounts-can-I-register-my-YubiKey-with-FIDO2>.
- [3] Passkeys. Google, 2022. <https://developers.google.com/identity/passkeys>.
- [4] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *IEEE Security & Privacy*, 2022.
- [5] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.
- [6] Sunpreet S Arora, Saikrishna Badrinarayanan, Srinivasan Raghuraman, Maliheh Shirvanian, Kim Wagner, and Gaven Watson. Avoiding lock outs: Proactive fido account recovery using managerless group signatures. *Cryptology ePrint Archive*, 2022.
- [7] Giuseppe Ateniese. Verifiable encryption of digital signatures and applications. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):1–20, 2004.
- [8] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ecDSA threshold signing. *Cryptology ePrint Archive*, 2020.
- [9] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.
- [10] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.
- [11] M Ben-Or, S Goldwasser, and A Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *STOC*, pages 1–10, 1988.
- [12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.
- [13] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *ACM STOC*. 1988.
- [14] Dan Boneh, Manu Drijvers, and Gregory Neven. BLS multi-signatures with public-key aggregation. <https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html>, Accessed 23 May 2022, March 2018.
- [15] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In *ASIACRYPT*, pages 331–345. Springer, 2000.
- [16] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, pages 126–144. Springer, 2003.
- [17] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *CCS*, pages 2075–2092, 2019.
- [18] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787, 2020.
- [19] Ran Canetti, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA. *Cryptology ePrint Archive*, 2020.
- [20] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS*, pages 1825–1842, 2017.
- [21] Melissa Chase, Apoorva Deshpande, Esha Ghosh, and Harjasleen Malvai. Seamless: Secure end-to-end encrypted messaging with less trust. In *CCS*, pages 1639–1656, 2019.
- [22] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *European Symposium on Research in Computer Security*, pages 654–673. Springer, 2020.
- [23] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bækvang Ostergård. Fast threshold ECDSA with honest majority. In *SCN*, pages 382–400. Springer, 2020.
- [24] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662. Springer, 2012.
- [25] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with Human-Memorable secrets. In *OSDI*, pages 1121–1138, 2020.
- [26] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. Accountable authentication with privacy protection: The larch system for universal login. *arXiv preprint arXiv:2305.19241*, 2023.

- [27] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.
- [28] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *EUROCRYPT*, pages 120–127. Springer, 1987.
- [29] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *EUROCRYPT*, pages 307–315. Springer, 1989.
- [30] Arkajit Dey and Stephen Weis. PseudoID: Enhancing privacy in federated login. In *HotPETS workshop*, 2010.
- [31] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *IEEE Security & Privacy*, pages 980–997. IEEE, 2018.
- [32] Corin Faife. Okta ends lapsus\$ hack investigation, says breach lasted just 25 minutes. *The Verge*, April 2022. <https://www.theverge.com/2022/4/20/23034360/okta-lapsus-hack-investigation-breach-25-minutes>.
- [33] Daniel Fett, Ralf Küsters, and Guido Schmitz. Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In *European Symposium on Research in Computer Security*, pages 43–65. Springer, 2015.
- [34] Daniel Fett, Ralf Küsters, and Guido Schmitz. Spresso: A secure, privacy-respecting single sign-on system for the web. In *CCS*, pages 1358–1369, 2015.
- [35] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT*, pages 186–194. Springer, 1986.
- [36] FIDO Alliance. FIDO alliance specifications: Overview. <https://fidoalliance.org/specifications/>, Accessed 20 May 2022.
- [37] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction: 23rd International Conference*, volume 8409, page 244, 2014.
- [38] Adam Gągol and Damian Straszak. Threshold ecdsa for decentralized asset custody. Technical report, Tech. rep., Cryptology ePrint Archive, Report 2020/498, 2020.
- [39] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645. Springer, 2013.
- [40] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *CCS*, pages 1179–1194, 2018.
- [41] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS*, pages 156–174. Springer, 2016.
- [42] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, pages 1069–1083, 2016.
- [43] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.
- [44] Trillian. <https://github.com/google/trillian>.
- [45] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326. Springer, 2016.
- [46] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 253–280. Springer, 2015.
- [47] Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. *Cryptology ePrint Archive*, 2022.
- [48] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In *EUROCRYPT*, 2022.
- [49] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. *Cryptology ePrint Archive*, 2021.
- [50] Chengqian Guo, Jingqiang Lin, Quanwei Cai, Fengjun Li, Qiong Xiao Wang, Ji Wu Jing, Bin Zhao, and Wei Wang. UPPRESSO: Untraceable and unlinkable privacy-preserving single sign-on services. *arXiv preprint arXiv:2110.10396*, 2021.
- [51] Sven Hammann, Ralf Sasse, and David Basin. Privacy-preserving OpenID connect. In *ASIACCS*, pages 277GS22–289, 2020.
- [52] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *IEEE Security & Privacy*, pages 285–303. IEEE, 2021.
- [53] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In *Proceedings of*

*the 2016 ACM on Workshop on Privacy in the Electronic Society*, pages 139–142, 2016.

- [54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.
- [55] Ivan Krstic. Behind the scenes with iOS security, 2016. <https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf>.
- [56] Krypton. kr-u2f. <https://github.com/kryptco/kr-u2f>, Accessed 17 May 2022.
- [57] SCIPR Lab. libsark. <https://github.com/scipr-lab/libsark>, Accessed 30 May 2022.
- [58] Leslie Lamport. The part-time parliament. In *TOCS*, pages 133–169, 1998.
- [59] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. <https://tools.ietf.org/html/rfc6962>.
- [60] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. SAVER: Snark-friendly, additively-homomorphic, and verifiable encryption and decryption with rerandomization. *Cryptology ePrint Archive*, 2019.
- [61] Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, pages 613–644. Springer, 2017.
- [62] Joshua Lund. Technology preview for secure value recovery, 2019. <https://signal.org/blog/secure-value-recovery/>.
- [63] Vadim Lyubashevsky and Gregory Neven. One-shot verifiable encryption from lattices. In *EUROCRYPT*, pages 293–323. Springer, 2017.
- [64] Philip MacKenzie and Michael K Reiter. Two-party generation of DSA signatures. In *CRYPTO*, pages 137–154. Springer, 2001.
- [65] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.
- [66] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.
- [67] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS# 1: RSA cryptography specifications version 2.2. *Internet Engineering Task Force, Request for Comments*, 8017:72, 2016.
- [68] D. M’Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.
- [69] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In *CCS*, pages 1717–1731, 2020.
- [70] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.
- [71] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, pages 238–252. IEEE, 2013.
- [72] Emma Roth. LastPass’ latest data breach exposed some customer information. *The Verge*, November 2022. <https://www.theverge.com/2022/11/30/23486902/lastpass-hackers-customer-information-breach>.
- [73] Adam Rowe. Study reveals average person has 100 passwords, November 2021. <https://tech.co/password-managers/how-many-passwords-average-person>.
- [74] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID connect core 1.0 incorporating errata set 1. [https://openid.net/specs/openid-connect-core-1\\_0.html](https://openid.net/specs/openid-connect-core-1_0.html), November 2014.
- [75] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [76] Nigel P Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, pages 342–366. Springer, 2019.
- [77] Markus Stadler. Publicly verifiable secret sharing. In *EUROCRYPT*, pages 190–199. Springer, 1996.
- [78] Akira Takahashi and Greg Zaverucha. Verifiable encryption from MPC-in-the-Head. *Cryptology ePrint Archive*, 2021.
- [79] Verizon. *DBIR Data Breach Investigations Report*, 2022 edition. <https://www.verizon.com/business/resources/T3cd/reports/dbir/2022-data-breach-investigations-report-dbir.pdf>.
- [80] W3C. Web authentication: An api for accessing public key credentials level 2, April 2021. <https://www.w3.org/TR/webauthn-2/>, Accessed 20 May 2022.
- [81] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. <https://developer.android.com/about/versions/pie/security/ckv-whitepaper>.



- [82] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and Abhi Shelat. Blind certificate authorities. In *IEEE Security & Privacy*, pages 1015–1032. IEEE, 2019.
- [83] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [84] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 21–37, 2017.
- [85] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In *CCS*, pages 558–573, 2021.
- [86] Shota Yamada, Nuttapong Attrapadung, Bagus Santoso, Jacob CN Schuldt, Goichiro Hanaoka, and Noboru Kunihiro. Verifiable predicate encryption and applications to CCA security and anonymous predicate authentication. In *PKC*, pages 243–261. Springer, 2012.
- [87] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.
- [88] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.
- [89] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *CCS*, pages 1919–1938, 2020.
- [90] Zhiyi Zhang, Michal Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Efficient and lightweight privacy-preserving single sign on. *PoPETS*, 2021(2):70–87, 2021.
- [91] ZoKrates. ZoKrates. <https://github.com/Zokrates/ZoKrates>, Accessed 30 May 2022.



# K9db: Privacy-Compliant Storage For Web Applications By Construction

Kinan Dak Albab    Ishan Sharma    Justus Adam    Benjamin Kilimnik    Aaron Jeyaraj  
Raj Paul    Artem Agvanian    Leonhard Spiegelberg    Malte Schwarzkopf  
*Brown University*

## Abstract

Data privacy laws like the EU’s GDPR grant users new rights, such as the right to request access to and deletion of their data. Manual compliance with these requests is error-prone and imposes costly burdens especially on smaller organizations, as non-compliance risks steep fines.

K9db is a new, MySQL-compatible database that complies with privacy laws by construction. The key idea is to make the data ownership and sharing semantics explicit in the storage system. This requires K9db to capture and enforce applications’ complex data ownership and sharing semantics, but in exchange simplifies privacy compliance. Using a small set of schema annotations, K9db infers storage organization, generates procedures for data retrieval and deletion, and reports compliance errors if an application risks violating the GDPR.

Our K9db prototype successfully expresses the data sharing semantics of real web applications, and guides developers to getting privacy compliance right. K9db also matches or exceeds the performance of existing storage systems, at the cost of a modest increase in state size.

## 1 Introduction

New privacy laws including the European Union’s General Data Protection Regulation (GDPR) [44], the California Consumer Privacy Act (CCPA) [10], and others [7, 17, 20, 56] seek to protect users’ rights to their data in web services. Many of these laws provide users with rights to issue subject access requests (SARs), including a *right to access*, which lets users request a copy of their data, and a *right to erasure*, which requires its deletion on request [51, 52]. Many also impose a mandate to store data securely. Compliance with these laws is important, as violations risk severe fines [9, 39–41].

Achieving compliance can be onerous and expensive, however, particularly for small and medium-size organizations. These organizations must write custom queries and track metadata to identify and extract data related to a user, and continuously maintain this infrastructure as services evolve. Even well-intentioned developers sometimes get it wrong: for example, the ownCloud collaboration platform [43], though it

claims GDPR compliance [42], retains a user’s activity log after account deletion. Retrofitting compliance onto existing systems is tricky, as it still requires manual work [2, 28] and may harm performance [51].

This paper explores an alternative system design that achieves privacy compliance *by construction*. Our key idea is to make data ownership a first-class citizen in the database system itself. K9db, our new database system, tracks sufficient information to know, for each row in the database, what user (or users) have rights to it. This allows K9db to infer correct procedures for data retrieval and deletion, so that the database itself can handle requests under the rights to access or erasure, freeing the application developer from having to write or maintain custom scripts to handle these requests. The ownership information also allows K9db to encrypt data with per-user keys, which helps meet, e.g., the GDPR’s “Protection by Design and Default” requirement, which can be satisfied by encrypting at-rest data [36, 44]. Finally, K9db uses ownership information to generate errors if the database schema or operations on database contents risk violating the GDPR.

To realize K9db, we had to address three challenges. First, K9db must understand and model the complex data ownership and sharing semantics of real applications. A user’s data may span many tables with transitive relationships, may be shared in complex and data-dependent ways, and may require partial redaction when returned or removed. Second, K9db must maintain and enforce compliance invariants matching these ownership semantics throughout application execution, and correctly respond to user access and deletion requests. Third, K9db should match the performance of today’s databases that lack infrastructure for data ownership tracking, and must be both compatible with existing applications and easy for application developers to adopt.

K9db’s design addresses these challenges as follows. First, K9db derives a *data ownership graph (DOG)* from a set of coarse-grained, declarative annotations on the database schema. Using a small number of primitives, the DOG models a wide range of complex data sharing relationships found in real-world applications. The DOG is central to K9db’s

storage organization, to its handling of users' access and erasure requests, and to K9db's ability to enforce privacy compliance. Second, K9db organizes data storage around data ownership to ensure that applications remain in compliance and handle access and deletion requests correctly by construction, without disrupting regular application operations. Third, K9db is a MySQL-compatible drop-in-replacement for existing databases, and requires few application changes beyond declarative schema annotations for normalized schemas. To accelerate complex queries, K9db provides an integrated, privacy-compliant in-memory cache based on materialized views. By integrating and managing materialized views, K9db provides the benefits of caching to applications, while relieving developers from ensuring compliance of cached data.

K9db structures the actual data storage as a set of user-specific logical "micro-databases" ( $\mu$ DBs), realized over a single physical RocksDB [33] store. Each user's  $\mu$ DB contains the data they own, and is encrypted with a user-specific key. K9db also helps developers use the system correctly by providing compliance-specific functionality not found in other databases. A new EXPLAIN COMPLIANCE SQL command gives the developer insight into the DOG and highlights possible schema annotation errors; and K9db supports *compliance transactions* that guard against dynamic compliance problems, such as data without an owner being left behind in the database. K9db provides ACID guarantees similar to those in default MySQL.

K9db provides out-of-the-box compliance for well-intentioned developers who want to comply with privacy laws, and helps developers avoid mistakes. We expect that fines for privacy violations (e.g., the greater than 4% of annual turnover or €25M for GDPR violations) discourage intentional misuse.

In summary, this paper makes the following contributions:

1. The data ownership graph (DOG) for modeling ownership in a database, specified with schema annotations.
2. K9db, a new database that enforces compliance-by-construction based on the DOG and a compliant, ownership-aware storage organization.
3. Mechanisms that, based on the DOG, warn developers if schema annotations are insufficient or if the database becomes non-compliant at runtime.
4. An evaluation of K9db, demonstrating that a database centered around first-class data ownership and compliance-by-construction is practical.

We evaluate K9db with scenarios based on the Lobsters web application [27], the ownCloud document sharing platform [43], and the Shuup e-commerce platform [53]. Our experiments show that K9db can express a wide variety of nuanced data sharing and ownership patterns found in these applications, and that K9db performs on-par with or better than MariaDB and the widely-used MariaDB/memcached stack when serving typical web application workloads.

K9db is open-source at <https://github.com/brownsys/K9db>.

## 2 Background and Related Work

### 2.1 Privacy Laws

Web services must comply with new privacy and data protection laws [7, 10, 17, 20, 44, 56]. Many of these laws have a comprehensive scope: e.g., the EU's GDPR applies to anyone who offers services to users physically in the EU and touches many aspects of web services [52]. In particular, most laws grant users rights over their data that require services to identify all data related to a user. The GDPR, for example, provides *Subject Access Requests* (SARs) that allow a "data subject" (i.e., an end user) to request a copy of their data (Right to Access, Art. 15), to request the deletion of their data (Right to Erasure, Art. 17), and to receive the data in a portable and machine-readable format (Right to Data Portability, Art. 20). Complying with SARs requires the service provider ("data controller" in GDPR terms) to identify the information related to a data subject. As the GDPR has become a model for other privacy laws, many have adopted similar SAR-like requirements. The California Consumer Privacy Act (CCPA), for example, gives consumers a right to request the "specific pieces of personal information [a business] has collected about the consumer" [10, §1789.110] and its deletion [10, §1789.105].

The GDPR and other laws also impose mandates for secure data handling, particularly encryption at rest [44, Arts. 25, 32, 10, §1798.150(a)(1)]. These mandates avoid prescribing particular technologies: e.g., the GDPR only requires that organizations take "appropriate technical measures" to secure personal data [44, Art. 32], giving freedom to meet the requirement in different ways. In practice, encrypting data at rest and deleting encryption keys (referred to as "crypto-shredding"), e.g., to make backups inaccessible, is widely considered a compliant approach [45].

This paper primarily focuses on technical infrastructure to ease compliance with SARs and the requirement for secure storage. Privacy laws also include other provisions that e.g., mandate user consent for processing and regulate data sharing with third parties. Our design is compatible with these requirements, but they are not the focus of this paper.

### 2.2 Complexity of Data Ownership

Compliance with SARs is difficult, both manually and in automated systems, because web services often have complex ownership and data sharing semantics. Identifying data associated with a particular user ("data subject") is challenging. In relational databases, these associations are expressed as foreign keys; but data in many tables link to data subjects transitively via one or more intermediate tables, rather than directly. Multiple data subjects can be associated with the same data (e.g., private messages), and sometimes this association is asymmetric and implies different rights for different data subjects (e.g., a teacher and a student). Finally, many-to-many relationships introduce dynamically changing associations

between data and a variable number of data subjects.

GDPR-like laws afford companies with some flexibility in handling SARs. Applications may keep data associated with the data subject (possibly in some anonymized form) after a deletion request due to legal or contractual obligations (e.g., tax laws) or public interest [44, Art. 17.3]. Data may also be retained depending on the purpose of its processing, including the interests of other users [44, Art 6.1, Art. 17.1(b)]. For example, Facebook’s privacy policy specifies that Facebook deletes the comments that a withdrawing data subject made, but not the private messages they sent to a friend, unless that friend also deletes them [16]. Thus, the compliance policy and exact handling of SARs are application and data dependent.

### 2.3 Existing Approaches to Privacy Compliance

Privacy compliance today requires application developers to write custom queries and maintain metadata to identify and track information related to each data subject [51]. The queries are tricky to get right and maintain as the application evolves. To address part of this burden, some large companies built bespoke GDPR metadata stores [13, §1] and dedicated frameworks for data deletion [14]. However, these frameworks only solve part of the problem, and most organizations lack the resources to build such systems themselves. Our work provides compliance within an off-the-shelf database.

Adaptations of existing database systems can go some way towards providing privacy compliance, but can come at a steep performance cost. For example, Shastri et al. found that secondary indexes and strict metadata tracking impose overheads up to  $5\times$  [51], leading to proposals to accelerate these operations in hardware [21]. SchengenDB [23] outlines a design that provides GDPR compliance, but relies on extensive metadata and conservative, coarse-grained enforcement, e.g., destroying entire VM clusters when a data subject deletes their account. Our work redesigns the database to make correct privacy compliance a first-class property [49], without sacrificing performance and with moderate overheads.

Other proposals have advocated restructuring web services to enforce users’ privacy rights, but face barriers to adoption. W5 [24], Oort [11], Blockstack [3], and Solid [29] decouple data storage from the web application and put data storage under user control. This approach allows for strong guarantees, but requires rewriting web applications, comes with restrictions (e.g., all application logic must run in JavaScript in the browser), and is incompatible with today’s advertising-based business model for web services. Data Capsules [58], Riverbed [57], and Zeph [8] let users specify individual privacy policies for their data in web services. Though powerful, custom policies do not solve the problem of identifying all information related to a user; and may limit possible operations (e.g., to those expressible as homomorphic additions). Our work provides by-construction compliance with subject access requests, but with a storage model and database interface that works for existing web applications.

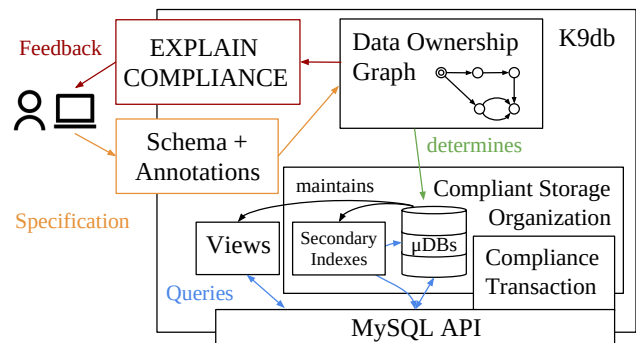


Figure 1: K9db provides privacy-compliant storage based on its data ownership graph, micro-databases ( $\mu$ DBs), and compliance helper mechanisms behind a MySQL interface.

## 3 K9db Overview

K9db is a relational database that makes data ownership an explicit first-class citizen. K9db targets typical web application workloads, which are dominated by reads and point lookup queries [18]. Its design goals are (i) to require few changes to application code, (ii) to capture and enforce the complex data ownership and sharing semantics of real-world applications, and (iii) to provide feedback that helps developers get privacy compliance right.

Figure 1 shows an overview of K9db’s components. K9db requires developers to extend their relational schema (i.e., `CREATE TABLE` statements) with a small set of annotations that encode data ownership and sharing semantics. The annotated schema acts as an application-specific compliance policy that specifies how K9db handles SARs. From these annotations, K9db builds its key abstraction, the *data ownership graph (DOG)* (§4). The DOG lets K9db determine, for every row in the database, who owns it and who has rights to it. K9db uses the DOG to satisfy data subjects’ SARs, to check that the database remains compliant after the application makes changes, and to warn the developer if their annotated schema and the compliance policy it encodes seem incomplete or contradictory.

Using information from the DOG, K9db organizes its storage in a user-centric way, storing each data subject’s data in their own logical “micro-database” ( $\mu$ DB), a shard of the actual database. This design ensures that K9db enforces the developer-provided compliance policy by construction, lets K9db encrypt each data subject’s data with a separate cryptographic key, and speeds up compliance-related enforcement and operations (§5). K9db maintains some additional secondary indices compared to a traditional SQL database, which help K9db efficiently resolve which  $\mu$ DBs store particular data. It also maintains materialized views that help simplify and accelerate execution of complex queries, while also providing an integrated, privacy-compliant in-memory cache (§6).

For normalized schemas, K9db requires little to no applica-



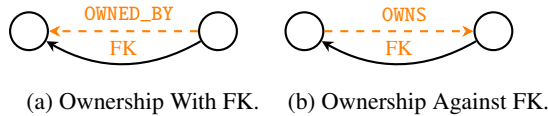


Figure 2: K9db’s annotations on foreign keys (FKs; orange) indicate the direction of data ownership (black edge) between two tables. Circles are tables.

Annotation	Example
DATA_SUBJECT	CREATE DATA_SUBJECT TABLE users (...)
$T_A(x)$ OWNED_BY $T_B(y)$	stories(author_id) OWNED_BY users(id)
$T_A(x)$ OWNS $T_B(y)$	member(gid) OWNS group(id)
$T_A(x)$ ACCESSED_BY $T_B(y)$	share(share_with) ACCESSED_BY user(id)
$T_A(x)$ ACCESSES $T_B(y)$	taggings(tag_id) ACCESSES tags(id)
ON DEL $T_A(x)$ {ANON (...)   DELETE_ROW}	ON DEL chat(receiver) ANON (receiver)
ON GET $T_A(x)$ {ANON (...)   DELETE_ROW}	ON GET review(paper_id) ANON (reviewer_id)

Figure 3: K9db’s table and column-level annotations. All annotations except DATA\_SUBJECT and ANON imply a foreign key from column  $x$  in table  $T_A$  to column  $y$  in  $T_B$ .

tion code changes, except that developers may need to wrap certain operations in a compliance transaction (§5.5). Developers can use K9db as a drop-in replacement for MySQL.

## 4 Modeling Data Ownership and Sharing

K9db aims to provide correct-by construction compliance with privacy laws, which requires K9db to respond to SARs correctly and enforce several invariants over the data and its storage. Correct compliance has two prongs: (i) a compliance policy that is consistent with the privacy law in question, and (ii) correct enforcement of this policy when handling both regular application operations and SARs.

The compliance policy is application-specific and depends on the relationships in the underlying data. For a single application, multiple policies may achieve compliance, and laws afford developers some flexibility in choosing a policy that matches their application’s semantics (§2.2).

In K9db, developers express their compliance policy using schema annotations, which K9db represents using the *data ownership graph* (DOG): a directed, acyclic multigraph whose vertices represent database tables, and whose edges represent ownership relationships between rows in the tables.

### 4.1 K9db’s Annotations

Developers use schema annotations on foreign keys to communicate their application’s data ownership and sharing se-

manantics to K9db. To communicate how the database represents human persons who have rights over data (“data subjects” in GDPR terms), the developer annotates one or more tables with the table-granularity DATA\_SUBJECT annotation.

Foreign keys (FKs) relate rows in tables to each other, and often imply ownership—consider e.g., a story pointing to its author. This is the simplest case: a story is owned by the row its FK value points to. K9db provides the OWNED\_BY keyword for developers to annotate such FKs (Figure 2a; §4.3 discusses transitive cases). But foreign keys may also point in the *opposite* direction of ownership, as is the case e.g., if a user table has a foreign key to their primary address. For such cases, K9db provides the OWNS annotation (Figure 2b).

In addition to ownership, an application may also have data that is owned by one data subject (who has the right to delete it when removing their account), but share it with others. For example, in the file sharing platform ownCloud [43], users want to share files with others, but when they remove their account have the file be removed for everyone. K9db lets developers express this with the ACCESSED\_BY annotation, and its dual for opposite-direction FKs, ACCESSES.

These annotations extend the semantics of foreign keys with compliance semantics, and while every annotation is applied to a foreign key, not every foreign key impacts ownership or needs to be annotated. For example, the foreign key connecting students in a university database with their declared majors carries no ownership information—the students do not own the majors—and should not be annotated.

K9db also provides table-level annotations that allow developers to specify that columns in a table need anonymizing in the context of SARs. This is important because a row may need redacting before returning the row as part of a right-to-access request (ON GET), or because a row may need to be retained in anonymized form (e.g., for tax compliance) after a data subject requests deletion of their data (ON DEL). Each anonymization annotation is associated with an ownership or access foreign key (i.e., an outgoing edge from the table in the DOG). This allows for different anonymization behavior depending on how the data subject who issued a SAR is connected to the data. For example, in the HotCRP conference review system [22], if a data subject who is both a reviewer and an author makes an access request, they should receive an unredacted copy of the reviews they wrote, but redacted, anonymized reviews for the papers they authored.

Figure 3 shows K9db’s complete set of schema annotations.

### 4.2 Expressing Developers’ Compliance Policies

We demonstrate how developers annotate their schema to express their desired compliance policy using two examples extracted from real applications: stories and messages in Lobsters (Figure 4), and file sharing in ownCloud (Figure 5).

In Lobsters, developers begin by annotating the users table, which records the application’s end-users, with DATA\_SUBJECT. A user may post several stories, and retains

```

1 CREATE DATA_SUBJECT TABLE users (id INT PRIMARY KEY, ...);
2 CREATE TABLE stories (
3   id INT PRIMARY KEY, title TEXT, ...
4   author INT NOT NULL OWNED_BY user(id)
5 );
6 CREATE TABLE tags (id INT PRIMARY KEY, tag TEXT, ...);
7 CREATE TABLE taggings (
8   id INT PRIMARY KEY,
9   story_id INT NOT NULL OWNED_BY stories(id),
10  tag_id INT NOT NULL ACCESSES tag(id)
11 );
12 CREATE TABLE messages (
13   id INT PRIMARY KEY, body text, ...
14   sender INT NOT NULL OWNED_BY user(id),
15   receiver INT NOT NULL OWNED_BY user(id),
16   ON DEL sender ANON (sender),
17   ON DEL receiver ANON (receiver)
18 );

```

Figure 4: Partial schema for Lobsters. Users own the stories they authored and their associations with tags. Messages are jointly owned by both sender and receiver.

```

1 CREATE DATA_SUBJECT TABLE user (id INT PRIMARY KEY, ...);
2 CREATE TABLE group (id INT PRIMARY KEY, title TEXT, ...);
3 CREATE TABLE member (
4   id INT PRIMARY KEY,
5   uid INT NOT NULL OWNED_BY user(id),
6   gid INT NOT NULL OWNS group(id)
7 );
8 CREATE TABLE share (
9   id INT PRIMARY KEY, ...
10  uid_owner INT NOT NULL OWNED_BY user(id),
11  share_with INT ACCESSED_BY user(id),
12  share_with_group INT ACCESSED_BY group(id)
13 );

```

Figure 5: Partial schema for ownCloud file sharing: users own their group membership, which owns the group; files have an owner and are shared with users who have access to them.

sole ownership of them: these stories must be retrieved or deleted when the user issues an SAR. Developers express this by annotating the author FK in `stories` with `OWNED_BY`. Lobsters also has a set of tags that represent discussion topics, e.g., `games` and `programming`. Users can assign tags to stories they posted, and have complete ownership of these associations. Developers express this by annotating the `story_id` column in `taggings` with `OWNED_BY`. This makes the story the owner of its taggings, transitively making the data subject who owns the story (i.e., its author) the owner of the associated taggings. But the tags themselves are not related to any data subject. Thus, developers annotate `tag_id` with `ACCESSES` (and not `OWNS`). As a result, a data subject receives a copy of their stories and associated tags when they request access, while disassociating tags from their stories and removing the stories themselves when requesting deletion.

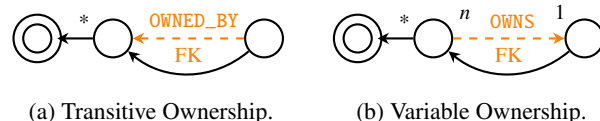


Figure 6: Tables can have transitive ownership relationships (\*: zero or more steps of indirection); if an edge follows a one-to-many or many-to-many relationship, it expresses variable ownership. Double circles indicate data subject tables.

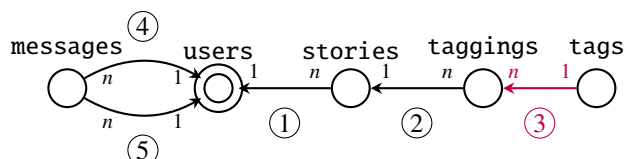


Figure 7: The DOG for stories and messages in Lobsters. Red indicates access-typed edges; 1 and  $n$  are cardinalities.

Similar to private messages in Facebook [16], messages in Lobsters are only deleted when both sender and receiver request deletion. Thus, developers annotate both sender and receiver with `OWNED_BY` (i.e., joint-ownership), along with anonymization annotations that instruct K9db to hide the identity of the associated withdrawing user in surviving messages. An alternative policy could require deleting a message as soon as one of the associated users is deleted. Developers can express this via an `ON DEL ... DELETE_ROW` annotation.

ownCloud’s data subjects are users in the user table, who can be members of a group (in the group table), as defined by the member association table. Users own their group memberships, so the developer annotates the `uid` column of member with `OWNED_BY`. The group and its associated resources are jointly owned by its members (ownCloud has no notion of group admins). Hence, the developer applies the `OWNS` annotation to the `gid` foreign key from member to group.

ownCloud’s share table contains records of users sharing files with others. This table specifies the file’s owner (i.e., its original creator) via the `uid_owner` column, which is a direct FK to the user table. The developer thus annotates this column with `OWNED_BY`. The `share_with` and `share_with_group` columns are also FKs that eventually lead to the user table, but indicate that the file is shared with (rather than owned by) these users. The developer therefore annotates them with `ACCESSED_BY`.

### 4.3 Data Ownership Graph

K9db builds the DOG from developers’ annotations by inserting DOG edges in the underlying FK direction for `OWNED_BY` and `ACCESSED_BY`, and against the FK direction for `OWNS` and `ACCESSES`. Thus, DOG edges always point towards a data subject table, unlike foreign keys.

When tables have a chain of annotated foreign keys, K9db

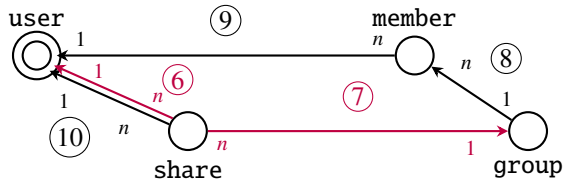


Figure 8: The DOG for ownCloud file sharing. Red edges are access-typed. Note the variable ownership (Fig. 6b) between member and group, as member rows are the group’s owners.

adds an edge to the DOG that establishes a *transitive* ownership relationship (Figure 6a). For example, in Lobsters (Figure 7), a story’s taggings have no direct references to the story’s author. Instead, they refer to their story ②, which in turn refers to the author ①. Therefore, edges in the DOG always represent a single step towards a data subject.

The DOG is a multi-graph because two tables can have multiple foreign keys between them. For example, in Lobsters the messages table has two foreign keys, one to the sender ④ and one to the receiver of a message ⑤. Since sender and receiver jointly own a private message—i.e., the message only disappears if both users delete their account—there are two annotated edges between messages and users.

Access annotations on foreign keys also add edges to the DOG, but these edges are access-typed and distinct from owner-typed edges. For example, in ownCloud (Figure 8) a file is accessible but not owned by users it is shared with, either directly ⑥ or via a group ⑦. Differentiating ownership and access edges is important for K9db to correctly handle access and deletion requests.

If the destination of a DOG edge can contain multiple rows corresponding to a single row in the source table, then that row can have multiple owners or accessors. The DOG edge ⑧ from ownCloud’s group to member is a one-to-many relationship, so a group may have many owners. This is an example of *variable ownership* (Figure 6b), as the number of owners varies depending on the data (i.e., depending on the rows in member). Similarly, DOG edges may also express *variable access*, e.g., a single tag in Lobsters may be accessed through many stories ③. This contrasts with the typical situation where the destination of a DOG edge is a primary key or unique column, making it a one-to-one or many-to-one relationship, both specifying a single owner (e.g., ⑨ and ⑩). K9db’s DOG metadata stores arity of relationships and K9db handles variable ownership and access appropriately.

#### 4.4 Helping Developers Get Annotations Right

EXPLAIN COMPLIANCE gives the developer information about the DOG, including heuristic warnings and suggestions about how it may be improved. K9db runs a simple heuristic over the schema to discover column names which indicate user data such as variations on “name”, “email” and “pass-

word”. If a table with such column names is not connected to a data subject in the DOG, K9db suggests to make it owned. This heuristic is most useful to discover missing data subjects, as their tables often contain columns with such names.

EXPLAIN COMPLIANCE also reports information that K9db derives from the DOG. For every table, it reports which data subject tables own it, and the paths through the DOG by which they own the table. This essentially shows the developer the closure over the DOG that K9db uses to handle SARs. EXPLAIN COMPLIANCE warns developers if a table is owned by many data subjects, e.g., if a DOG path contains multiple variable ownership edges, which can result in multiplicatively many owners. Such liberal sharing is rare in practice and likely the result of a schema or annotation mistake.

#### 4.5 Data Ownership Graph Properties

The DOG is *well-formed* if any path through it terminates at a data subject table. K9db rejects any schema that results in a DOG that is not well-formed.

Although the DOG is a graph of tables, its edges represent relations between rows in the source and destination tables based on the values of the underlying FK columns. Each DOG edge maps to a *relation* between rows in the two tables, where matching rows in the destination table own (or access) the rows in the source table. Intuitively, this relation can be evaluated as a query over the destination table, which yields exactly the owning row (or rows, in the case of variable ownership). Well-formedness guarantees that the transitive closure of these relations terminates at data subject tables.

Several key properties follow from this. First, if no matching rows exist in any destination table when evaluating the relations along *all* of the table’s outgoing ownership edges, data is orphaned (i.e., has no owner). This gives rise to the necessary (but insufficient<sup>1</sup>) *no orphaned data* compliance condition: any row in a database table connected to the DOG must resolve to  $\geq 1$  owning data subjects. Second, the transitive closure of relations corresponding to ownership edges in the DOG, starting from any row, identifies the set of data subjects that own this row. Third, the DOG’s reverse transitive closure starting from a row in a data subject table yields:

1. the rows shared with and owned by that data subject, if considering accessor-typed and owner-typed edges; or
2. the rows owned by that data subject, if considering only owner-typed edges.

The former set corresponds to the data that needs returning from a right-to-access request, and the latter identifies the data that needs deleting for a right-to-erasure request, provided no other owners exist.

### 5 Compliant by Construction Storage

In principle, the DOG and its relations are sufficient to identify a data subject’s data, and one could imagine adding it as a metadata layer over an existing database. But in practice,

<sup>1</sup>Sufficiency would require the *correct* owners, not just any owner.

compliance is more complex. Although the DOG identifies all data owned by a data subject, K9db needs to take the correct actions on this data. For example, K9db must avoid prematurely deleting jointly-owned data, and deletion must cover backups outside the live database. K9db must also have efficient ways to decide if a given database operation will break compliance, e.g., by violating the *no orphaned data* invariant, something that the DOG alone fails to provide.

K9db therefore introduces ownership as a first-class notion into the storage layer. This makes it simple for K9db to handle SARs, and to enforce invariants that must hold for compliance. Specifically, K9db's storage layer is organized around per-data subject logical "micro-databases" ( $\mu$ DBs), such that each  $\mu$ DB contains all of its data subject's owned data. For jointly-owned data, K9db stores copies of that data in the  $\mu$ DB of every data subject that owns it.

This design has several advantages. First, it ensures data deletion is correct relative to the DOG. When a data subject requests to delete their data, it is sufficient to delete their  $\mu$ DB. Data shared with other data subjects survives as copies in the other  $\mu$ DBs. Second, this design provides an easy way to check whether data is orphaned, as such data can only exist outside of all data subjects'  $\mu$ DBs. Third, this design lets K9db use a per-data subject key to encrypt data in each  $\mu$ DB. This simplifies deletion alongside external and replicated backups of the data, as deleting the owner's key makes all backups and copies inaccessible (i.e., "crypto-shredding").

### 5.1 Storage Layout and Logical $\mu$ DBs

K9db determines the  $\mu$ DBs to store each row in using the DOG. In a well-formed DOG, every table reaches at least one data subject table via its outgoing ownership edges. K9db splits the contents of such a table into different  $\mu$ DBs, each of which contains the rows owned by a particular data subject, and encrypts them with a key specific to that data subject. A table also includes an orphaned data section that may be used temporarily within sequences of operations (§5.5). A data subject's  $\mu$ DB therefore includes rows from every table that stores data owned by them. Note that even though  $\mu$ DBs store physical copies of rows that have multiple owners, they are a logical abstraction and realized over a single underlying physical datastore (e.g., RocksDB in our prototype).

Viewing the datastore as a whole, a previously single row in a table may now be multiple rows due to copies being stored in each owner's  $\mu$ DB. The value of the primary key of that row refers to all these copies. Internally, K9db identifies the different copies using a pairing of the data subject identifier (the value of its primary key in the data subject table) and the value of the primary key in the row.

K9db maintains on-disk secondary indexes separate from tables and  $\mu$ DBs, which K9db uses to execute queries efficiently. K9db creates an on-disk index for each unique and foreign key column and for the primary key. K9db on-disk indexes differ from traditional database indexes in two key

aspects: they map keys to ( *$\mu$ DB identifier, primary key*), and they point to all copies of any jointly-owned row that match the indexed key. K9db creates a special index for the primary key column(s) of owned tables, which maps the PK value to data subject identifiers that own the corresponding row.

K9db stores tables unconnected to the DOG in the same way as other databases. Such tables contain data that is not owned by any data subject, e.g., all available tags in Lobsters or all majors in a university database, and thus are outside any  $\mu$ DB. Note that this is distinct from orphaned data, which are rows without owners in tables that *are* connected to the DOG.

### 5.2 $\mu$ DB Integrity

The storage layer maintains an important invariant for compliance,  *$\mu$ DB completeness*: data owned by a data subject is exactly identical to the data stored in their  $\mu$ DB.

To maintain  $\mu$ DB completeness, K9db must identify the  $\mu$ DBs to insert new data into, and correctly apply application updates that change who owns rows. Changes to the data in a table may have cascading effects on who owns data in dependent tables connected to this table via some ownership path in the DOG. For example, changes to the `member` table in `ownCloud` affect who owns records in the `group` table. K9db utilizes the DOG to handle these situations correctly.

**Inserting Data.** When K9db receives an INSERT statement, it uses the DOG to identify the owners of this data. In particular, K9db analyzes the outgoing edges from the DOG vertex for the affected table. For a direct ownership edge, the data subject identifier is already present in the new row in the form of a foreign key. K9db determines this by introspection on the new row and without querying other tables. If an edge indirectly leads to the data subject table, identifying the owner becomes more complex. K9db can find the owner(s) by querying the database along the transitive edges between the table and the data subject. But such a query may be expensive—for example, the DOG for the Shuup e-commerce application [53] contains a chain of five edges from the `payments` table to the owning data subject. Instead, K9db memoizes the query by building and maintaining in-memory *ownership indexes*, which essentially provide "shortcut" relations over the DOG that point directly to the owning data subjects. In practice, K9db can often avoid or reuse ownership indexes (§6.1).

**Cascading Updates.** INSERT, UPDATE, or DELETE statements may have cascading effects on the ownership of records in dependent tables. After applying such statements to their target table, K9db identifies dependent tables from the DOG. It then queries the rows in each dependent table that match the updated row. K9db moves or copies the matched rows between  $\mu$ DBs appropriately, and cascades again into any further dependent tables. K9db requires no additional indexes to perform this matching efficiently, as it can rely on standard on-disk indexes over foreign keys' source and destination columns. In many cases, K9db avoids cascades via optimizations based on foreign key integrity (§6.1).



### 5.3 Handling Subject Access Requests

K9db needs to handle two types of SARs: the *right to access* and the *right to erasure*. K9db handles both with a similar high level procedure: (i) K9db traverses the DOG to identify all tables and edges connected to the data subject; (ii) K9db finds the data owned by the data subject in their  $\mu$ DB; (iii) K9db locates data accessed, but not owned, by the data subject in other  $\mu$ DBs; and (iv) K9db performs anonymization as specified by the developers in the schema.

For either type of request, K9db identifies the data subject’s data by following paths in the DOG, starting from the data subject table, and moving against incoming edges. A path that consists solely of ownership edges signifies data owned by the data subject, while paths that contain one or more access edges reflect accessed data. K9db locates the relevant rows in a table before moving on to any dependent tables. For every incoming edge, K9db uses the rows it located in the parent table to identify dependent rows in the dependent table. K9db finds these either in the same  $\mu$ DB for ownership paths, or in other  $\mu$ DBs using on-disk indexes for access paths.

After traversing an edge and retrieving data in its source table, K9db selects the anonymization annotations in the schema that apply to that edge. The anonymization annotations specify the columns to anonymize (e.g., the sender of a chat message). For access requests, K9db anonymizes retrieved rows before sending them back to the client. On deletion requests, K9db removes the data subject’s  $\mu$ DB from the database, and anonymizes any remaining copies of the data, which it locates in other  $\mu$ DBs using on-disk indexes.

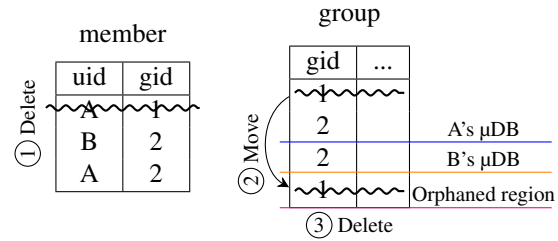
### 5.4 Atomicity, Consistency, Isolation, and Durability

A single SQL statement may result in several underlying operations over K9db’s storage, as it may update rows in several  $\mu$ DBs or cascade over dependent tables. It is critical for compliance that we ensure that these updates are all ACID, to avoid data races that could lead to a non compliant state (e.g., by creating orphaned data, or breaking the  $\mu$ DB completeness invariant). Therefore, K9db executes every SQL statement as a single statement ACID transaction (similar to MySQL). This includes all underlying operations over any  $\mu$ DBs and all updates to on-disk secondary indices or the integrated in-memory cache (§6.2). Our prototype does not support general multi-statement SQL transactions yet (see §7).

K9db guarantees that concurrent SQL statements have *repeatable reads* isolation, which is the default in MySQL. Any weaker isolation level is insufficient for compliance, as it cannot guarantee that K9db’s compliance invariants hold in the presence of concurrent updates.

### 5.5 Compliance Transactions

An application may itself perform operations that risk violating compliance. Consider the example from ownCloud shown in Figure 9: ① the application deletes user “A”’s membership in group 1, of which “A” is the last remaining



START COMPLIANCE TX

① DELETE FROM member WHERE uid=A AND gid=1;

② K9db applies cascading effect;

gid=1 is orphaned

COMPLIANCE BROKEN

③ DELETE FROM group WHERE gid=1; COMPLIANCE RESTORED

COMMIT COMPLIANCE TX

Figure 9: K9db’s *compliance transactions* help developers check that the database is in a compliant state after multiple operations (here, (1) deleting the last owner of a group, and (3) then deleting the group). Without a compliance TX, K9db would report an error instead of applying step (2).

member. This deletion from *member* has a cascading effect on the dependent *group* table. Since the group with *gid* 1 no longer has any owners, K9db ② moves it into the table’s orphaned data region. This breaks compliance, as it violates the DOG’s *no orphaned data* invariant. A correct application must now perform some operation that restores compliance, e.g., by deleting group 1 in a separate SQL operation, which ③ removes the orphaned row, restoring the invariant.

K9db supports this pattern with the idea of a *compliance transaction* (CTX). A CTX wraps a set of operations that may temporarily violate compliance, but commits only if the database is back to a compliant state at the commit point. Within a CTX, K9db stores orphaned data in orphaned regions attached to each table. On subsequent operations that reintroduce owners for this data, K9db migrates the rows from the orphaned regions to the corresponding  $\mu$ DBs; if deleted, K9db removes the data. At the end of a CTX, K9db ensures that every record moved to the orphaned region during the CTX has an owner again (or was deleted), and produces an error to the developer otherwise.

Finally, K9db forbids statements that write to the orphaned region unless they are part of a CTX. In particular, step ① in Figure 9 will error unless contained in a CTX. This means that developers need to modify applications that contain such patterns to use CTXs when necessary. Requiring such limited modification is desirable, as disallowing compliance-breaking changes outside of CTX helps developers identify issues and forces them to fix buggy and in-compliant applications. For example, K9db would reject a buggy version of ownCloud that does not clean up groups with no members ③. Introducing a CTX allows an application to have benign temporary in-compliance; if K9db instead required applications to only

perform operations that move the database between compliant states (e.g., deleting groups before deleting their last member), it would likely require more substantial rewrites.

CTX are different from regular SQL transactions, which serve to ensure consistency under concurrent execution. CTX are lightweight and required for compliance, while SQL transactions are expensive and web applications often (but not always) avoid them. In a privacy-compliant database with SQL transactions, each such transaction must also be a CTX.

## 6 Query Execution

When K9db executes a query, it must identify the  $\mu$ DBs affected to locate the relevant rows. Depending on the operation, this may involve finding one or all copies of shared rows.

Queries that refer to a single table, such as DELETE and UPDATE statements, and most SELECT queries issued by web applications (e.g., point lookups), run directly against K9db's  $\mu$ DBs with the aid of on-disk indexes. K9db analyzes the columns that appear in the WHERE condition of the query, and selects the index that matches the most columns. Like other databases, K9db finds all the rows that may match the query using the selected index, and then filters these rows with any remaining columns. If no index matches, K9db runs a scan over the table. Developers may create additional indexes using CREATE INDEX, similar to traditional databases.

When data has multiple owners, an index may refer to multiple copies of the same row. For DELETE and UPDATE, K9db atomically operates over all these copies, ensuring that all copies are consistent. K9db may need to remove or add some of the affected rows from/to  $\mu$ DBs, and may need to cascade into dependent tables as described in §5.2. For SELECT queries, K9db identifies a single copy of each matching row and skips any remaining index entries for other copies. This avoids overheads for deduplicating copies of the row.

K9db serves some complex SELECT queries from materialized view, described in §6.2.

### 6.1 Optimizations

K9db speeds up query execution and reduces its memory footprint with a set of optimizations designed to avoid deep cascades and to reduce the number of in-memory ownership indexes (§5.2) required. Some of these optimizations rely on *foreign key integrity*, which K9db enforces (like many other databases) to prevent application operations that result in dangling foreign keys. With FK integrity, rows cannot be inserted into a table if they contain references to non-existent rows in a destination table, and rows in the destination table cannot be deleted as long as source table rows refer to them.

**Avoiding Cascades.** K9db needs to cascade into dependent tables along incoming DOG edges to update dependent rows affected by a write (i.e., those owned by a modified row). But FK integrity guarantees that no such rows exist when K9db handles INSERT and DELETE queries to a table  $T$  that is the destination of a FK from a dependent table. This lets

K9db skip cascades along  $T$ 's incoming DOG edges if the edge is in FK direction; otherwise, K9db must cascade.

**Ownership Indexes.** K9db relies on two techniques to reduce the number of ownership indexes. First, multiple incoming DOG edges that require an ownership index and point to the same column of a table (usually the primary key) may reuse the same index. Second, K9db omits ownership indexes for edges in the DOG that correspond to OWNS annotations, such as the edge from `group` to `member` in `ownCloud`. These edges point in opposite direction to the underlying foreign key. FK integrity ensures that a row must exist at the source of such an edge (e.g., `group`) before any rows referring to it can be inserted to the destination table (e.g., `member`). Hence, K9db always inserts new rows from the source table into the orphaned region, and defers moving them to the correct  $\mu$ DB to future inserts into destination tables in the DOG (which must cascade), as discussed in §5.5. These optimizations, for example, help K9db create only one ownership index for `Lobsters` (which gets re-used three times), and avoid the need for any ownership indexes in `ownCloud`.

**Queries With Inlined Owners.** SQL Statements sometimes directly refer to the owner of their target rows, e.g., by constraining a foreign key that corresponds to an ownership edge in the DOG. Queries that fit this pattern are common in the web applications: e.g., in `Lobsters`, `SELECT * FROM stories WHERE author = ?` selects stories by their author, which is an annotated foreign key to `users`. K9db detects this situation by statically analyzing the WHERE condition and determines the relevant  $\mu$ DB without an on-disk index lookup.

### 6.2 Materialized Views

K9db serves complex SELECT queries, such as joins, aggregations, and those that reorder data, from materialized views. This design makes sense for two reasons. First, it is simple and avoids the need to engineer a sophisticated query planner that understands the nuances of ownership and indexes to efficiently execute these queries over K9db's  $\mu$ DBs. Second, developers often cache the results of complex SELECT queries in external systems (e.g., `memcached`). Privacy compliance while using an external cache requires setting appropriate expiration policies for the cache [59, §4.5] or explicit invalidation of cache entries related to a data subject if they request deletion of their data. This can be painful for developers and may require manually tracking metadata, e.g., when caching aggregates over many data subjects' data. Instead, K9db provides an integrated privacy-compliant cache using materialized views.

When K9db receives a complex SELECT query for the first time, it creates a materialized view and serves further instances of the query from it, until the view is removed or times out. K9db keeps the materialized views up to date via an incremental, streaming dataflow computation triggered by writes to  $\mu$ DBs, as well as  $\mu$ DB deletion. This makes inserts, updates, and deletes more expensive, but speeds up reads.

K9db updates the materialized views atomically prior to acknowledging the corresponding operation to the client. This, along with our storage layer, ensures *repeatable reads* isolation for concurrent operations whether cached or not.

K9db’s ownership indexes are special-case materialized views, maintained with the same dataflow infrastructure.

## 7 Implementation

Our K9db prototype consists of 35k lines of C++, 500 lines of Rust, and 2k lines of Java. It relies on RocksDB for  $\mu$ DB storage, on Apache Calcite [6] for query planning, and on libsodium [15] for encryption. Our implementation is similar to the MyRocks MariaDB storage engine [30], but extends it with compliance and  $\mu$ DB capabilities.

**MySQL Compatibility Layer.** K9db exposes a MySQL binary protocol interface, so unmodified applications can treat K9db as a MySQL server. The interface to K9db’s materialized views is primarily through prepared SQL statements: when an application registers a prepared statement, K9db creates a view if necessary and serves future executions of the prepared statement from it. Developers can also create additional views manually.

**Storage.** K9db relies on RocksDB for persistent data storage. Each table in the schema is a RocksDB column family. Rows in K9db are keyed by a combination of their owner and primary key, to uniquely identify each owner’s copy of a row. Our prototype stores rows ordered by their owner identifier, and uses that identifier as a RocksDB prefix. This allows it to extract and delete  $\mu$ DBs using RocksDB prefix iterators. Our prototype creates and maintains on-disk indexes as RocksDB column families, and formats their content to allow writes to retrieve all the copies of a row, and reads to retrieve a single arbitrary copy, skipping the rest. Like MySQL, K9db creates indexes for primary, unique, and foreign keys.

**Encryption at Rest.** K9db uses hardware-accelerated AES256-GCM to encrypt all data in a  $\mu$ DB with the key of its owner. The key ( $\mu$ DB identifier, primary key) associated with every row is encrypted deterministically with a global key to allow consistent lookup. This has leakage, but is sufficient to satisfy the GDPR’s “security of processing” requirement (Art. 32), which is often interpreted to require encryption of data at rest [4]. It is possible to use blind indexes [5] which also allow consistent lookup but reduce leakage. K9db’s design is independent of the particular encryption scheme used, and can benefit from future advances in searchable encryption. Information in materialized views and secondary indexes remains unencrypted, but K9db deletes it when deleting a user’s data. K9db destroys the decryption key when a user removes their account, making any remaining backups inaccessible.

**ACID.** K9db executes each application SQL statement in a RocksDB transaction, which is based on row-level locking. This includes all updates to secondary indices (similar to MyRocks) and all  $\mu$ DBs and cascade operations. As in MyRocks, K9db serves reads from a consistent RocksDB

snapshot. K9db also updates all relevant materialized views prior to committing. Unlike MyRocks, K9db enforces foreign key integrity and appropriately locks FK targets during execution. Overall, this ensures that concurrent SQL statements are atomic and consistent with *repeatable reads* isolation, which is the default in MySQL and MyRocks.

**View Updates.** K9db’s materialized view updates follow a standard design akin to differential dataflow [32, 35] and Noria [18]. Each table in the schema is associated with an input vertex in the dataflow graph, and when K9db performs updates to a table, it injects the updates into its dataflow input vertex. The dataflow processes the updates through a sequence of operators to derive an incremental update to the materialized view (or secondary index), and applies this update. Dataflow operators are stateless (e.g., projections, filters, unions) or stateful (e.g., joins, aggregations). K9db’s materialized views are indexed for ordered and unordered lookups.

**Limitations.** Our prototype lacks support for general, multi-statement SQL transactions. These are rare in web applications, and can be supported using existing RocksDB primitives and techniques for versioned dataflow processing [31, 35]. While our prototype does not yet support schema changes, RocksDB is schema-oblivious, and our prototype’s storage layer could be extended to support schema changes with some engineering effort, using similar techniques to MyRocks. Finally, K9db’s dataflow graph operators sometimes store copies of a record; by using a record pool, our prototype’s memory footprint could be reduced.

## 8 Evaluation

We evaluate K9db with three applications, Lobsters [27], ownCloud [43], and Shuup [53]. We ask three questions:

1. What is K9db’s impact on end-to-end application performance? (§8.1)
2. What is the impact of K9db’s design features on performance? (§8.2)
3. What effort by application developers does using K9db require? (§8.3)

We run experiments on a Google Cloud n2-standard-16 VM, storing databases on a local SSD. Our baselines use MariaDB v10.6.5 (a MySQL fork) with the RocksDB-based MyRocks storage engine, and memcached v1.6.10.

### 8.1 Application Performance

We start by analyzing K9db’s performance with two applications: Lobsters and ownCloud.

#### 8.1.1 Lobsters

Lobsters ([lobste.rs](http://lobste.rs)) is an open-source discussion board, similar to Reddit. Lobsters currently lacks GDPR compliance [26], and has a schema that consists of 19 tables, which store posts, comments, nested replies, upvotes, invitations and other information. We annotated this schema for K9db with three DATA\_SUBJECT tables, 14 OWNED\_BY, one ACCESSES, and



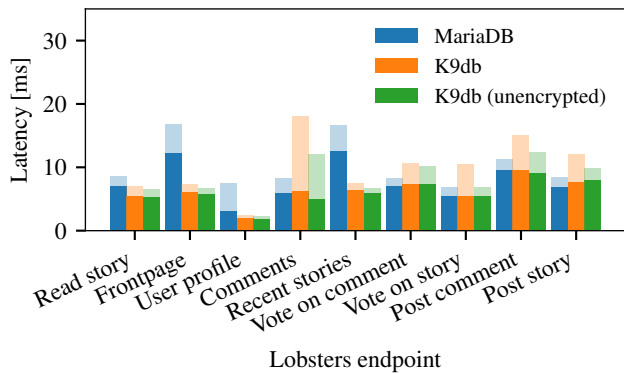


Figure 10: K9db matches or beats MariaDB’s median (solid) and 95<sup>th</sup> percentile (shaded) latency on the Lobsters workload, and encryption has low overheads except on the “Comments” endpoint, which reads thousands of rows in the tail.

two anonymization annotations (details in §8.3). We use an existing open-source, open-loop benchmark for Lobsters based on public workload statistics [19]. The benchmark models ten endpoints in the Lobsters webapp that correspond to different pages and each issue between six and fifteen SQL queries, most of which are reads. We load the database with data that models the current production Lobsters deployment (15k users, 100k stories, 313k comments, and 416k votes) [19]. K9db therefore maintains 15k logical  $\mu$ DBs in this experiment. We compare MariaDB, and K9db with and without data encryption. (Encryption with per-user keys isn’t possible in the MariaDB baseline.) Lobsters on most requests runs an expensive query to determine the user’s recently read stories. This query joins four tables, including the (large) stories and comments tables. This query is slow in MariaDB ( $\approx 30$ ms) and dominates its latency for all endpoints, while K9db serves this query from a materialized view. To make the comparison fair, we remove the expensive query in the MariaDB baseline. A good result for K9db would show latencies comparable to MariaDB for all endpoints, and a low overhead for encryption.

Figure 10 shows the results. Endpoints that mostly read (on the left) benefit from K9db’s materialized views and are up to  $2.1\times$  faster than in MariaDB, but endpoints with many writes (on the right) are comparable in both systems. This makes sense, as K9db performs similar work to MariaDB, except that some read queries are served from materialized views, and writes need to be encrypted and must update any corresponding views. K9db without encryption is on-par with K9db in most endpoints. For the “Comments” endpoint, K9db is  $2.1\times$  slower than MariaDB and  $1.5\times$  slower than K9db without encryption in the 95<sup>th</sup> percentile. This happens when the endpoint retrieves comments and votes on a popular story from the database, which requires K9db to decrypt thousands of records. Developers could manually add materialized views in K9db to speed up this endpoint, at the cost of additional memory. Other endpoints read fewer rows or rely on (unen-

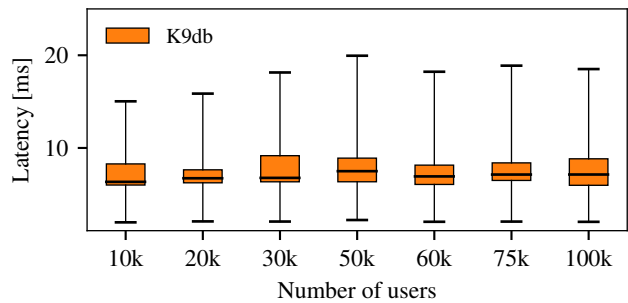


Figure 11: K9db’s 95<sup>th</sup> percentile latency on the Lobsters workload remains stable as the number of users (and thus,  $\mu$ DBs) increases. Each bar shows a distribution of endpoint latencies.

rypted) materialized views. This shows that K9db achieves good performance for a practical web application, and that encryption has acceptable cost. All further experiments show results for K9db with encryption enabled.

We chose the load in this experiment to saturate the hardware for the MariaDB baseline ( $\approx 760$  pages/second, which results in 10k queries/second) and used the same load for K9db. K9db supports a up to a  $4.8\times$  higher load without latency degradation, thanks to its caching for complex queries via materialized views; we compare to a caching MariaDB+memcached baseline below.

**Subject Access Requests.** We now measure the time required by K9db to satisfy SARs. We issue an access and a deletion request for each of the top 1000 users with most data in the database, and run these requests sequentially through K9db SARs API. Performance of SARs is secondary as they are rare operations and can be executed asynchronously. A good result shows that K9db handles SARs correctly (which it does by construction) and within reasonable time. In our experiment, K9db on average takes 1 ms to retrieve and 45 ms to delete the correct data for a user.

**Scalability.** We designed K9db to have performance independent of the number of  $\mu$ DBs. We confirm this using the Lobsters benchmark with different numbers of users. Adding users increases the number of  $\mu$ DBs and the amount of data in the database, but keeps the average amount of data per user constant. A good result for K9db would show latencies remaining constant as the number of users grows.

Figure 11 shows the results as box-and-whisker plots over the nine endpoints (i.e., the bottom and top whiskers are the fastest and slowest endpoints, respectively). K9db’s latency remains constant as the number of users—and, consequently,  $\mu$ DBs—grows, because K9db satisfies queries either from  $\mu$ DBs directly, via indexes, or from materialized views. These results confirm that K9db’s logical  $\mu$ DB partitioning is practical for applications with large numbers of users.

**Comparison to Caching Baseline.** In the previous experiment, K9db had an unfair advantage over MariaDB: it serves some data from materialized views, while Mari-



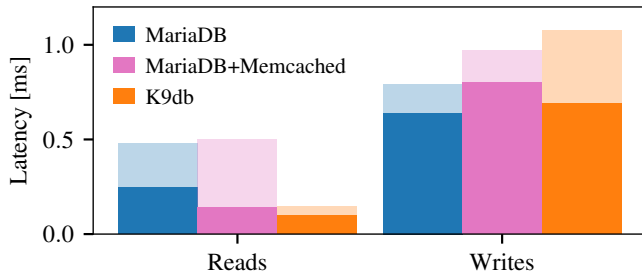


Figure 12: K9db matches MariaDB+memcached on a common Lobsters query (solid: median; shaded: 95<sup>th</sup>-ile).

aDB recomputes queries every time. We now use one common query from Lobsters to compare three setups: (i) standalone MariaDB; (ii) MariaDB with an in-memory cache (“MariaDB+Memcached”); and (iii) K9db. The MariaDB+Memcached setup is a demand-filled cache [37]: writes invalidate the cached query result in memcached, and the next read re-runs the query against the database when it misses in memcached. In K9db, writes update views via its dataflow graph. We generate a skewed workload with a Zipfian distribution ( $s = 0.6$ ) where 95% of requests in the benchmark read the details of a random story and its vote count, and 5% of requests insert new votes. A good result for K9db would show competitive read performance with memcached and low overheads on write processing (since K9db does more work on writes); and MariaDB+Memcached and K9db would show lower latencies than MariaDB alone.

Our results are in Figure 12. For reads, MariaDB+Memcached and K9db are on par in the median, but K9db has a lower 95<sup>th</sup> percentile latency as K9db updates the cache via streaming dataflow, while MariaDB+Memcached queries the database on a read miss. All systems perform similarly on writes, as this query requires little dataflow update work in K9db and the caching baseline must make an extra RPC to invalidate memcached.

**Memory Overhead.** K9db’s materialized views and ownership indexes add memory overhead compared to a traditional database. We measure this cost and compare it to a caching setup with memcached. We consider a setup that caches query results that developers would typically store in memcached, such as the output of expensive joins and aggregates. These queries are identical to the ones that K9db caches using materialized views. The experiment caches query results with the query parameters (? in prepared statements) as the key, and the concatenated records as the value. K9db stores additional in-memory data for internal dataflow state and ownership indexes. A good result for K9db would therefore show moderate overheads compared to MariaDB+Memcached.

The Lobsters database is 61 MB on disk, and a typical memcached caching approach stores an additional 97 MB of in-memory state. K9db’s memory footprint is 197 MB ( $3.3 \times$  DB size, and  $2 \times$  memcached’s footprint), which includes

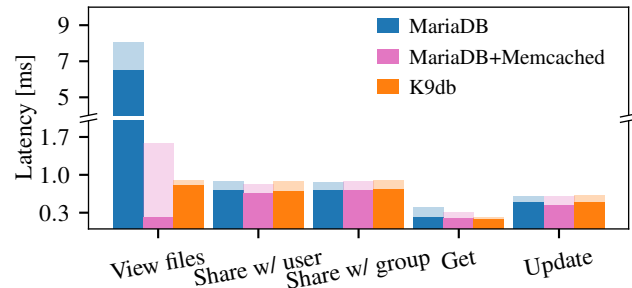


Figure 13: K9db matches the baseline setups’ performance on the ownCloud workload (solid: median; shaded 95<sup>th</sup>-ile).

6.5MB for the stories ownership index, and 56 MB for caching the expensive query we removed from MariaDB (without this query, K9db’s overhead is  $2.4 \times$  DB size/ $1.5 \times$  memcached). The overhead comes from K9db’s dataflow state, which allows K9db to incrementally update materialized views.

### 8.1.2 ownCloud

ownCloud is a popular open-source application that allows users to upload files and share them with other users [43]. Recall ownCloud’s schema (Figure 5): each file has a single owner—the original uploader—but users can share files with other users and with groups. Files shared with a group are accessible to all members of the group—i.e., a many-to-many relationship between users and files (a pattern absent in Lobsters). We measure five common queries: (i) listing the files a user can access (“view files”); (ii) sharing a file with another user (“share with user”); (iii) sharing a file with a group (“share with group”); (iv) retrieving a file using its primary key (“Get”); and (v) updating the retrieved file (“Update”). Our setup uses 100k users who each own three documents; each document is shared uniformly at random with three users and two groups; and each group has five members. Our workload is 95% read and 5% writes, equally split among the two types of sharing and file updates. Reads and writes target users drawn from a Zipf distribution ( $s = 0.6$ ). We batch ten reads and measure the per-request latency for the same setups as in the previous experiment. A good result for K9db would show comparable read latency to MariaDB+Memcached and low overheads on writes.

Figure 13 shows the results. “View files”, which returns all files shared with a user (directly or via a group), involves five tables and three joins, which MariaDB executes on every read. MariaDB+Memcached and K9db serve precomputed results from memory instead, which is fast. The 95<sup>th</sup> percentile for MariaDB+Memcached suffers because it queries MariaDB on a cache miss, which occurs when a query retrieves files of user(s) invalidated by a previous write. K9db is fast and stable because it updates the views via dataflow on writes. All systems perform similarly for the two share queries—a good result for K9db, as it also updates views.

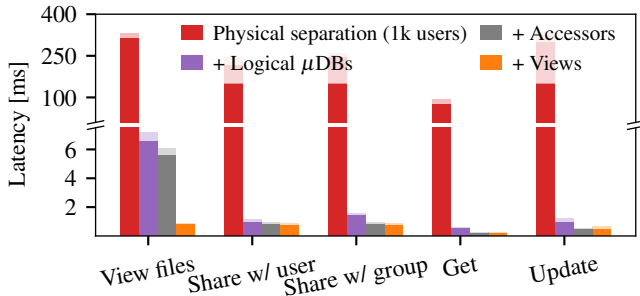


Figure 14: K9db matches the baseline setups’ performance on the ownCloud workload (solid: median; shaded 95<sup>th</sup>-ile).

## 8.2 K9db Design Drill-Down

To evaluate the impact of design decisions central to K9db, we run ownCloud workload from the previous experiment against versions of K9db that disable key components. We start with K9db set up to naïvely store every  $\mu$ DB in its own database (without cross- $\mu$ DB indexes); without support for accessor edges in the DOG; and without materialized views (i.e., queries always run over data in RocksDB). This guarantees strict separation of user’s data, a solution sometimes adopted for GDPR compliance in practice [46, 47], although this lacks support for shared data (accessors) or anonymization. We then add separation into logical  $\mu$ DBs (“+ Logical  $\mu$ DBs”), accessor support (“+ Accessors”), and materialized views (“+ Views”). A good result would show that these features improve K9db’s performance.

Figure 14 shows the results. The naïve  $\mu$ DB design is very slow because every query that K9db cannot statically resolve to the affected  $\mu$ DBs requires scanning all  $\mu$ DBs; we only ran this setup with 1k users (vs. 100k for the others). Making  $\mu$ DBs a logical abstraction much improves performance, justifying our design choice. Accessor-typed DOG edges are important for expressivity: without them, ownCloud would be restricted to a policy where users jointly own shared files. In addition, accessor support reduces the number of copies stored and the fan-out of writes, which slightly reduces query latency. Finally, materialized views improve latency of the “View files” query by 5 $\times$ , as the results are cached in memory. Since the view update is cheap, writes do not suffer much overhead. The runtime of “View files” without no views is comparable to the runtime of the same query in MariaDB (Figure 13). This illustrates that views are beneficial, but not essential to good performance in K9db.

## 8.3 Schema Annotation Effort

To understand the developer effort K9db’s schema annotations require, we now consider annotations for three applications (Lobsters, ownCloud, and Shuup [53]) in detail,

**Lobsters.** The Lobsters schema contains 19 tables. To use K9db, we had to annotate the schemas for eight tables. Three tables (users, invitations, and

invitation\_requests) contain data subjects. We annotated two FKs in each of hats, messages, and moderations with OWNED\_BY to model joint ownership. We annotated 8 other tables with a single OWNED\_BY. For example, votes has multiple foreign keys that lead to the users table (one direct, two indirect), and thus requires a single OWNED\_BY annotation to disambiguate and ensure votes are stored with the voter, rather than the author of the story or comment voted on. Finally, we used one ACCESSES in taggings, and two anonymization rules in messages, as shown in Figure 4.

**ownCloud.** ownCloud’s schema has 51 tables. We focused on the file sharing core, which consists of six tables and has the most complex relationships. In addition to the annotations in Figure 5, we added an OWNS annotation to the FK in the share table that points to the corresponding file in the file table (omitted from Figure 5 for brevity).

ownCloud’s original schema “overloads” the share\_with column to either hold a user or a group ID, and includes a share\_type column to distinguish these cases. K9db could support such de-normalized schema with more advanced conditional annotations; for our benchmarks, we modified the schema to track users and groups in separate columns.

**Shuup.** Shuup [53] is an open source e-commerce platform and supports customers with accounts, guests who do not have accounts, and shop owners, all of whom have GDPR rights. The Shuup code lets users request their account to be anonymized, but retains information for tax compliance, e.g., payment data, customers’ countries of residence, and tax ID numbers, a form of data retention provided for in the GDPR.

Shuup provides GDPR compliance via a manually-implemented module with 4k lines of Python code (2.7k lines of implementation and 1.3k lines of tests), developed in 137 commits over three years. At the time of writing, Shuup’s anonymization behavior is inconsistent; it only anonymizes default shipping and billing addresses, but retains previous addresses in cleartext in the mutable\_address table [55]. Moreover, downloading data for a user is not supported [54].

We implemented Shuup’s anonymization policy in K9db using all annotations (Figure 3) over 17 of Shuup’s 278 tables. We annotate personcontact with DATA\_SUBJECT. This table stores natural persons, and has FKs to their contact information (in contact) and their logins (in auth\_user) if they have accounts. Thus, personcontact contains users with and without accounts, i.e., guests. Using K9db, Shuup correctly anonymizes data, lets users download the data and fixes the bug of not anonymizing previous default addresses.

Shuup’s schema has several tables that might correspond to data subjects. K9db’s EXPLAIN COMPLIANCE helps developers understand that they need to annotate personcontact. An in-compliant (but plausible) alternative would be to annotate auth\_user, the login details table. This results in contact being unconnected to the DOG, as there are no foreign keys to auth\_user. The personcontact table has such a foreign key, but it is nullable (e.g., for guests who lack

Application	Tables	Data Subject	Owner	Access	Anon
Commento [12]	12	3	8	1	3
ghChat [1]	6	1	7	2	4
HotCRP [22]	26	2	15	10	7
Instagram clone [50]	19	1	18	1	0
Mouthful [25]	3	1	1	0	0
Schnack [48]	5	1	1	0	0
Socify [34]	19	1	10	0	0

Figure 15: K9db requires few DATA\_SUBJECT, ownership (OWNED\_BY and OWNS), access (ACCESSED\_BY, ACCESSES), and ANON annotations to support real web applications.

accounts), and thus some of its rows will be stored in  $\mu$ DBs and others in the orphaned region. EXPLAIN COMPLIANCE helps developers identify and rectify these issues:

- 
- ```

1 Table "contact": GLOBAL
2 [Compliance Warning] Column "email" suggests personal
  data, but the table is not connected to any owners.
3 Table "personcontact": in  $\mu$ DB for auth_user.id
4 [Compliance Warning] Table has owners, but nullable
  foreign key may prevent correct deletion of data.

```
- 

A developer might also annotate `contact` with DATA\_SUBJECT, but that table includes entries for customers and companies. Annotating it makes companies into data subjects, which duplicates company-related tables across  $\mu$ DBs. EXPLAIN COMPLIANCE also alerts developers to this.

**Other Applications.** Our schema annotations were sufficient to express reasonable compliance policies for seven additional applications (Figure 15). We briefly highlight several interesting patterns in these applications.

In `ghChat` [1], a chat application for GitHub, and the `Instagram clone` [50], a group is owned exclusively by its admin and accessed by its members. This is unlike `ownCloud`, which lacks group admins and has members jointly own the group.

`Mouthful` [25] is a commenting service that embeds in a host application (e.g., a blog) to allow users to comment on the host content (e.g., a blog post). `Mouthful` has no notion of users; instead, the host application provides a string that represents the user identity alongside the comment they posted. We added a DATA\_SUBJECT table to store user identifiers, and created a FK constraint from the `Comment` table’s `author` column to it.

Finally, the `HotCRP` [22] review system associates data subjects to papers via a many-to-many `PaperConflict` table. The table has a `conflictType` column that specifies the relationship, such as “co-author” or “institutional conflict”. While this schema is normalized in the traditional SQL sense, it is not normalized for ownership: rows with the co-author type signify ownership, while other conflict types do not imply any ownership or access rights over the paper. We resolved

this by adding a new `PaperAuthors` table that only stores authorship associations, and refer to papers from it using `OWNS`. We reserve the existing `PaperConflict` to record other conflict types with an un-annotated reference to papers.

**Migrating Applications to K9db.** We identify some common challenges when migrating applications to K9db. First, annotating an application schema requires knowledge of the application functionality and its compliance policy, but also summarizes the policy in an easy-to-maintain way alongside the schema. Many web applications also lack explicit FK constraints in their schema; developers must identify the columns that act as implicit FKs and annotate them if needed.

Second, applications often have schemas that are not normalized in the traditional SQL sense (e.g., `ownCloud`’s `share_with`) or with regards to ownership (e.g., `HotCRP`’s `PaperConflict`). Developers must normalize these schemas by introducing new columns or tables, and apply the corresponding changes to the application code. K9db could support such schemas via new annotations that condition on other columns, but this would complicate the annotation language and DOG model. Instead, K9db guides developers to good, normalized schema designs.

Finally, applications with variable ownership (e.g., `ownCloud`, `Shuup`, `HotCRP`) often have endpoints that temporarily orphan data. Developers must wrap such endpoints in compliance transactions in order to use K9db. This modification is relatively unobtrusive, and K9db can be configured to automatically wrap sessions in a CTX. This alleviates the need to manually introduce CTX to applications that open new sessions for each endpoint or sequence of operations, but is not suitable for applications with long-lived sessions.

## 9 Conclusion

K9db is a new database system that achieves compliance with the requirements of privacy laws by construction.

K9db models data ownership to capture the ownership patterns of real world applications, and handles requests for access and deletion correctly. K9db matches or exceeds the performance of a widely-used database and manual caching setup, and supports the privacy requirements of real-world applications. K9db is open-source and available at <https://github.com/brownsys/K9db>.

## Acknowledgements

We are grateful to Deniz Altınbüken, Hannah Gross, Frans Kaashoek, Franco Solleza, Lillian Tsai, and the ETOS group at Brown for helpful feedback on drafts of this paper. Feedback from the anonymous reviewers and our shepherd, Nat-acha Crooks, greatly improved the paper. We also thank Vedant Gupta, Mithi Jethwa, and Colton Rusch for contributions to K9db’s implementation.

This research was supported by NSF awards CNS-2045170 and DGE-2039354, by a Google Research Scholar Award, and by a gift from VMware.

## References

- [1] aermin. *ghChat (react version)*. URL: <https://github.com/aermin/ghChat> (visited on 05/02/2021).
- [2] Archita Agarwal, Marilyn George, Aaron Jeyaraj, and Malte Schwarzkopf. “Retrofitting GDPR Compliance onto Legacy Databases”. In: *Proceedings of the VLDB Endowment* 15 (Dec. 2021).
- [3] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. “Blockstack: A Global Naming and Storage System Secured by Blockchains”. In: *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, Colorado, USA, June 2016, pages 181–194.
- [4] Amazon Web Services. *Navigating GDPR Compliance on AWS: Encrypt Data at Rest*. URL: <https://docs.aws.amazon.com/whitepapers/latest/navigating-gdpr-compliance/encrypt-data-at-rest.html> (visited on 05/05/2021).
- [5] Scott Arciszewski. *Building Searchable Encrypted Databases with PHP and SQL*. May 2017. URL: <https://paragonie.com/blog/2017/05/building-searchable-encrypted-databases-with-php-and-sql>.
- [6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. “Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources”. In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, Texas, USA, 2018, 221–230.
- [7] National Congress of Brazil. *Lei Geral de Proteção de Dados [Brazilian General Data Protection Law]*. English translation by Ronaldo Lemos, Daniel Douek, Sofia Lima Franco, Ramon Alberto dos Santos and Natalia Langenegger. URL: [https://iapp.org/media/pdf/resource\\_center/Brazilian\\_General\\_Data\\_Protection\\_Law.pdf](https://iapp.org/media/pdf/resource_center/Brazilian_General_Data_Protection_Law.pdf) (visited on 06/11/2020).
- [8] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. “Zeph: Cryptographic Enforcement of End-to-End Data Privacy”. In: *Proceedings of the 15<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 387–404.
- [9] California Attorney General. *Privacy Enforcement Actions*. URL: <https://oag.ca.gov/privacy/privacy-enforcement-actions> (visited on 05/06/2021).
- [10] California Legislature. *The California Consumer Privacy Act of 2018*. June 2018. URL: [https://leginfo.ca.gov/faces/billTextClient.xhtml?bill\\_id=201720180AB375](https://leginfo.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375).
- [11] Tej Chajed, Jon Gjengset, M. Frans Kaashoek, James Mickens, Robert Morris, and Nickolai Zeldovich. *Oort: User-Centric Cloud Storage with Global Queries*. Technical report MIT-CSAIL-TR-2016-015. MIT Computer Science and Artificial Intelligence Laboratory, Dec. 2016.
- [12] Adhityaa Chandrasekar. *Commento*. URL: <https://github.com/adtac/commento> (visited on 05/02/2021).
- [13] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. “Unearthing inter-job dependencies for better cluster scheduling”. In: *Proceedings of the 14<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020, pages 1205–1223.
- [14] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papiannidis. “DELF: Safeguarding deletion correctness in Online Social Networks”. In: *Proceedings of the 29<sup>th</sup> USENIX Security Symposium (USENIX Security)*. Banff, Canada, Aug. 2020.
- [15] Frank Denis. *The Sodium cryptography library*. 2013. URL: <https://download.libsodium.org/doc/>.
- [16] Facebook. *Permanently Delete Your Facebook Account*. URL: [https://www.facebook.com/help/224562897555674?helpref=faq\\_content](https://www.facebook.com/help/224562897555674?helpref=faq_content) (visited on 05/21/2023).
- [17] Thailand Government Gazette. *Personal Data Protection Act*. Unofficial English translation. URL: <https://thainetizen.org/wp-content/uploads/2019/11/thailand-personal-data-protection-act-2019-en.pdf> (visited on 06/11/2020).
- [18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. “Noria: dynamic, partially-stateful data-flow for high-performance web applications”. In: *Proceedings of the 13<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 213–231.
- [19] Peter Bhat Harkins. *Lobsters access pattern statistics for research purposes*. Mar. 2018. URL: [https://lobsters.com/s/cqnz15/lobsters\\_access\\_pattern\\_statistics\\_for#c\\_hj0r1b](https://lobsters.com/s/cqnz15/lobsters_access_pattern_statistics_for#c_hj0r1b) (visited on 03/12/2018).



- [20] PRS Legislative Research India. *The Personal Data Protection Bill, 2019*. URL: <https://www.prsindia.org/billtrack/personal-data-protection-bill-2019> (visited on 06/11/2020).
- [21] Zsolt István, Soujanya Ponnappalli, and Vijay Chidambaram. “Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach!” In: *Proceedings of the VLDB Endowment* 14.7 (Mar. 2021), pages 1167–1174.
- [22] Eddie Kohler. *HotCRP conference review software*. URL: <https://github.com/kohler/hotcrp> (visited on 07/22/2020).
- [23] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. “SchengeDB: A Data Protection Database Proposal”. In: *Proceedings of the 2019 VLDB Workshop Towards Poly-stores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019, pages 24–38.
- [24] Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, and Michael Walfish. “A World Wide Web Without Walls”. In: *Proceedings of the 6<sup>th</sup> Workshop on Hot Topics in Networks (HotNets)*. Atlanta, Georgia, USA, Nov. 2007.
- [25] Viktoras Kuznecovas. *Mouthful*. URL: <https://github.com/vkuznecovas/mouthful> (visited on 05/02/2021).
- [26] Lobste.rs. *Privacy: Lobsters*. URL: <https://lobste.rs/privacy> (visited on 05/01/2021).
- [27] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: <https://lobste.rs> (visited on 03/02/2018).
- [28] Connor Luckett, Andrew Crotty, Alex Galakatos, and Ugur Cetintemel. “Odlaw: A Tool for Retroactive GDPR Compliance”. In: *Proceedings of the 37<sup>th</sup> IEEE International Conference on Data Engineering (ICDE)*. Chania, Greece, Apr. 2021.
- [29] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. “A Demonstration of the Solid Platform for Social Web Applications”. In: *Proceedings of the 25<sup>th</sup> International Conference Companion on World Wide Web (WWW)*. Montréal, Québec, Canada, 2016, pages 223–226.
- [30] MariaDB. *MyRocks – MariaDB Knowledge Base*. URL: <https://mariadb.com/kb/en/myrocks/> (visited on 12/06/2022).
- [31] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Mothy Roscoe. “Shared Arrangements: practical inter-query sharing for streaming dataflows”. In: *Proceedings of the VLDB Endowment* 13.10 (June 2020), pages 1793–1806.
- [32] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. “Differential dataflow”. In: *Proceedings of the 6<sup>th</sup> Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.
- [33] Meta Platforms, Inc. *RocksDB: A persistent key-value store for fast storage environments*. URL: <http://rocksdb.org/> (visited on 12/10/2022).
- [34] Sudharsanan Muralidharan. *Socify: open source social network using Ruby on Rails*. URL: <https://github.com/scafffeinate/socify> (visited on 05/02/2021).
- [35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. “Naiad: a timely dataflow system”. In: *Proceedings of the 24<sup>th</sup> ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.
- [36] European Network and Information Security Agency. *Privacy and data protection by design: from policy to engineering*. 2015. URL: <https://data.europa.eu/doi/10.2824/38623>.
- [37] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. “Scaling Memcache at Facebook”. In: *Proceedings of the 10<sup>th</sup> USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398.
- [38] Noria Contributors. *Noria Lobsters benchmark*. 2020. URL: <https://github.com/mit-pdos/noria/tree/3edd3ad55d2564493f7456d27abb41abf0169def/applications/lobsters>.
- [39] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-008*. URL: [https://gdprhub.eu/index.php?title=CNIL\\_-\\_SAN-2020-008](https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-008) (visited on 05/06/2021).
- [40] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-018, Nestor SAS*. URL: [https://gdprhub.eu/index.php?title=CNIL\\_-\\_SAN-2020-018](https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-018) (visited on 05/06/2021).

- [41] NOYB: European Center for Digital Rights. *GDPRHub: GPDDP 9485681, Vodafone Italia*. URL: [https://gdprhub.eu/index.php?title=Garante\\_per\\_la\\_protezione\\_dei\\_dati\\_personali\\_-\\_9485681](https://gdprhub.eu/index.php?title=Garante_per_la_protezione_dei_dati_personali_-_9485681) (visited on 05/06/2021).
- [42] ownCloud GmbH. *GDPR compliant cloud storage*. URL: <https://owncloud.com/gdpr> (visited on 12/01/2021).
- [43] ownCloud GmbH. *owncloud – share files and folders, easy and secure*. URL: <https://owncloud.com> (visited on 12/01/2021).
- [44] “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the European Union* L119 (May 2016), pages 1–88.
- [45] Brent Robinson. *Crypto shredding: How it can solve modern data retention challenges*. 2019. URL: <https://medium.com/@brentrobinson5/crypto-shredding-how-it-can-solve-modern-data-retention-challenges-da874b01745b>.
- [46] Alexander Rubin. *40 million tables in MySQL 8.0 with ZFS*. URL: <https://www.percona.com/blog/2018/09/03/40-million-tables-in-mysql-8-0-with-zfs/> (visited on 05/03/2021).
- [47] Alexander Rubin. *One Million Tables in MySQL 8.0*. URL: <https://www.percona.com/blog/2017/10/01/one-million-tables-mysql-8-0/> (visited on 05/03/2021).
- [48] schnack! *schnack.js*. URL: <https://github.com/schn4ck/schnack> (visited on 05/02/2021).
- [49] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. “GDPR Compliance by Construction”. In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019.
- [50] Faiyaz Shaikh. *React-Instagram-Clone-2.0*. URL: <https://github.com/yTakkar/React-Instagram-Clone-2.0> (visited on 05/02/2021).
- [51] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. “Understanding and Benchmarking the Impact of GDPR on Database Systems”. In: *Proceedings of the VLDB Endowment* 13.7 (Mar. 2020), pages 1064–1077.
- [52] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. “How Design, Architecture, and Operation of Modern Systems Conflict with GDPR”. In: *Proceedings of the 11<sup>th</sup> USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. July 2019.
- [53] Shuup Commerce, Inc. *Shuup Open-Source E-Commerce Platform*. URL: <https://github.com/shuup/shuup> (visited on 12/05/2021).
- [54] Shuup Contributors. *GDPR - Download Data button doesn't return any data*. URL: <https://github.com/shuup/shuup/issues/2614> (visited on 12/13/2021).
- [55] Shuup Contributors. *GDPR - shuup\_mutaddress rows not anonymized*. URL: <https://github.com/shuup/shuup/issues/2612> (visited on 12/13/2021).
- [56] Griffin Thorne. *GDPR Meets its Match ... in China*. July 2019. URL: <https://www.chinalawblog.com/2019/07/gdpr-meets-its-match-in-china.html> (visited on 06/04/2020).
- [57] Frank Wang, Ronny Ko, and James Mickens. “Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services”. In: *Proceedings of the 16<sup>th</sup> USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Feb. 2019, pages 615–630.
- [58] Lun Wang, Joseph P. Near, Neel Somani, Peng Gao, Andrew Low, David Dao, and Dawn Song. “Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations”. In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019, pages 3–23.
- [59] Juncheng Yang, Yao Yue, and K. V. Rashmi. “A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter”. In: *ACM Transactions on Storage* 17.3 (2021).

## A Artifact Appendix

### Abstract

Our open source artifact contains our prototype implementation of K9db. It also includes the harnesses and scripts for running and plotting the experiments described in this paper.

Our prototype provides a MySQL-compatible interface layer, which applications and developers can use to issue SQL statements and queries to and retrieve their results. Our prototype is compatible with the standard MySQL connectors and drivers for several languages, including C++, Rust, and Java. It is also compatible with the command line MySQL and MariaDB clients.

## Scope

Our prototype serves as a demonstration of the following:

1. The application scenarios described in the paper work with K9db and its schema annotations.
2. K9db's system design and guarantees can be realized with a familiar MySQL-compatible interface suitable for web applications.
3. The performance of compliant-by-construction databases is comparable to traditional databases, such as MariaDB.

## Contents

**K9db.** The artifact includes our prototype implementation and its MySQL-compatibility layer. The artifact contains instructions for building, running, and using this K9db.

**Application Harnesses.** The artifact includes harnesses for Lobsters, a Reddit-like discussion board (§8.1.1), and own-Cloud (§8.1.2), a file sharing application. The harnesses create the database schema and load the database with data; they also execute loads with representative queries, and measure the time required to process them. We used these harnesses to evaluate our prototype and the baselines shown in our experiments. The Lobsters harness is a pre-existing open source harness that we adapted to work with our prototype [38].

**Documentation.** The artifact wiki on GitHub contains a tutorial on using K9db and its schema annotations. The artifact also includes unit and end-to-end tests that validate that our prototype handles application SQL operations correctly and provides correct compliance with SARs.

## Hosting

Our artifact is hosted on GitHub at <https://github.com/brownsys/K9db>. The version of the repository corresponding to this paper is available at <https://github.com/brownsys/K9db/releases/tag/osdi2023>, with commit hash *df2bcdffa05f70f508fad95a11e2a6de8a7efe14*. The corresponding wiki commit hash is *c720b085ca34edc16246f296991e623a29933f9b*.

## Requirements

We developed our prototype on x86-64 machines running Ubuntu 20.04 and 22.04. We provide a Docker container that includes the necessary software dependencies. We ran our experiments on Google Cloud using `n2-standard-16` machines with a local SSD.

# Encrypted Databases Made Secure Yet Maintainable

Mingyu Li<sup>1,2,3</sup> Xuyang Zhao<sup>1,3</sup> Le Chen<sup>1,3</sup> Cheng Tan<sup>†</sup> Huorong Li<sup>\*</sup> Sheng Wang<sup>\*</sup>  
 Zeyu Mi<sup>1,3</sup> Yubin Xia<sup>1,2,3</sup> Feifei Li<sup>\*</sup> Haibo Chen<sup>1,3</sup>

<sup>1</sup>Shanghai Jiao Tong University    <sup>2</sup>Shanghai AI Laboratory    <sup>†</sup>Northeastern University    <sup>\*</sup>Alibaba Group  
<sup>3</sup>Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

## Abstract

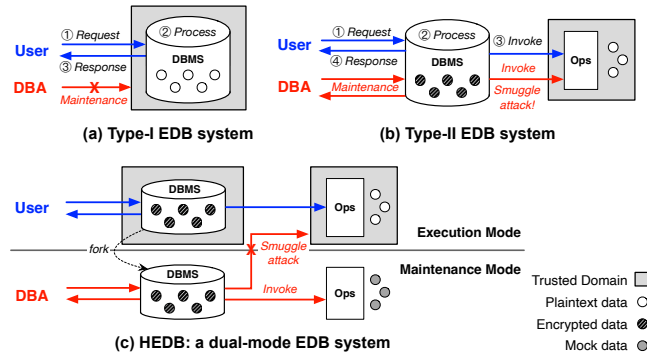
State-of-the-art encrypted databases (EDBs) can be divided into two types: one that protects the whole DBMS engine in a trusted domain, and one that protects only operators that support queries over encrypted data. Both types have limitations when dealing with malicious database administrators (DBAs). The first type either exposes the data to DBAs or makes maintenance operations difficult if the DBA role is eliminated. The second type is vulnerable to abuse of the operator interfaces; in particular, we devise a smuggle attack that enables DBAs to secretly and effectively access data.

We introduce HEDB, which prevents smuggle attacks and preserves database maintainability. HEDB uses a dual-mode EDB design based on our analysis of DBA maintenance tasks. Execution Mode handles user queries by isolating DBAs from operators to prevent smuggle attacks, while Maintenance Mode enables DBMS maintenance and operator troubleshooting through *authenticated replay* and *anonymized replay*, respectively. Our evaluation shows that HEDB blocks smuggle attacks and supports common maintenance tasks with 5.88% runtime cost and 9.26% storage cost.

## 1 Introduction

With approximately 60 ZB of data stored in database systems [6], much of which is sensitive, data breaches pose one of the most serious security threats today, causing an average loss of \$4.35 million per incident [4]. To protect against external attacks, database security features such as role-based access control and encryption at rest have become *de facto* standards. However, these features are not effective at preventing attacks from malicious insiders, who create new internal threats. This is especially true for Database as a Service (DBaaS) scenarios, where cloud platform operators and database administrators (DBA) have full access to the database engine and customer data. To address this threat, several encrypted database (EDB) systems have been proposed by academia [15, 17, 26, 42, 44, 48] and industry [14, 30, 50].

Despite a broad spectrum of prior efforts [14, 15, 17, 26, 30, 42, 44, 48, 50], EDB systems with (i) full-SQL functionality, (ii) maintainability and (iii) strong security have remained an unsolved problem for the past decade. State-of-the-art EDB systems can be largely categorized into two types: (1)



**Figure 1:** Existing EDB systems can be categorized into two types: Type-I lacks maintenance and Type-II lacks interface security. HEDB leverages a dual-mode design to support both.

a monolithic EDB design that isolates the whole database engine in a trusted domain, and (2) a plug-and-play EDB design that leverages protected operators to process user secrets. We name them Type-I and Type-II for brief, as depicted in Figure 1. Both types reuse existing database engines, inheriting (almost) all features of modern databases such as SQL execution and ACID transactions.

For Type-I EDBs [17, 44, 45], a system operator or DBA can only monitor an end-to-end secure channel between an isolated database engine and a remote user. However, the conventional role of DBAs conflicts with customer privacy. Consider a maintenance task that DBAs help to diagnose a database misconfiguration bug [40]. After curious DBAs log into the database server, they can read whatever user data of interest because of their high privilege. Notably, eliminating the role of DBAs from the database engine requires non-trivial engineering efforts. Furthermore, excluding DBAs will give up the benefits of their expertise in managing, optimizing and diagnosing the outsourced databases.

Type-II EDBs [14, 15, 30, 42, 43, 48, 50] typically use database extensions to enable various primitive operators over encrypted data. Operators include arithmetics, comparisons, string searching, etc. The primary advantage of Type-II is that operators have a small trusted computing base (TCB) compared to Type-I. Furthermore, the low complexity of the operator’s codebase also makes it easy to develop and simple to vet. Most importantly, Type-II surpasses Type-I in terms of



maintainability, as DBAs can connect to the database server, examine query plans, attach powerful profilers and debuggers, and collect crash dumps without concerns of data breaches, since user data is always encrypted. Type-II EDBs are therefore well adopted by cloud database vendors [14, 30, 50]. Regarding data privacy, Type-II leaks information such as ordering and frequency, which may compromise sensitive columns with the aid of sophisticated background knowledge [27–29, 32, 39].

Even worse, under existing database access control, an adversarial DBA can arbitrarily invoke Type-II EDB’s operator interfaces. As Type-II exposes various operators, DBAs can exploit a sequence of carefully constructed invocations to recover the victim’s sensitive data [16]. We devise an efficient and stealthy attack, named *smuggle attacks*, which applies to all basic encrypted types (i.e., numerics, time, text) and can recover 100% data items of 35,243 health records within 2 minutes with no prior knowledge. Conceptually, smuggle attacks is similar to Iago attack [21], as both abuse interfaces. But unlike Iago, defending smuggle attacks is more challenging because it does not tamper with the correctness of invocation results. Hence, we opt for a new approach to defeating smuggle attacks, while retaining Type-II’s advantages of DBA maintainability.

**Our proposal: HEDB<sup>1</sup>.** HEDB is a new EDB design that can provide interface security (namely, smuggle attacks-resilient) and maintainability. HEDB’s design is based on two insights: (a) without authenticated access, interface security cannot be achieved, and (b) in most cases, accessing plaintext secret data is not essential to EDB maintenance. Hence HEDB introduces two modes: *Execution Mode* where operators authenticate valid requests for user queries (defending against smuggle attacks), and *Maintenance Mode* where mock data is used during DBA maintenance (minimizing privacy leakages). In Execution Mode, HEDB adopts Type-II’s design by decoupling the DBMS and operators, and protects them using two trusted domains with an authenticated channel. When switching to Maintenance Mode, HEDB forks a new DBMS instance from the protected DBMS to an unprotected domain, and feeds operators (also in the unprotected domain) with mock data. Figure 1 overviews this process.

This dual-mode EDB design is non-trivial and has several technical challenges. First, switching EDB components between modes requires execution environment reconstruction for maintenance purposes. Second, too accurate maintenance may help DBAs infer secret data easily, while simply using fake data hinders maintenance. Third, after maintenance, there should be a secure way to apply hotfixes to the protected DBMS instance, without invoking any new attack surfaces.

To overcome the above challenges, HEDB introduces several key techniques. To allow DBAs to inspect the stateful DBMS, HEDB employs DBMS-located VM fork across two

hypervisors using existing hardware (i.e., ARMv8.4 S-EL2). For execution environment reconstruction, HEDB relies on record-and-replay. HEDB records the operator invocation trace in Execution Mode, and proposes *authenticated replay* to reproduce DBMS issues in Maintenance Mode. To preserve buggy control flows and protect user data privacy at the same time, HEDB proposes *anonymized replay*, which employs concolic execution to capture path constraints, translates data masking rules also into constraints, and exploits constraint solving for operator troubleshooting in Maintenance Mode. Finally, HEDB uses *maintenance templates* to securely apply hotfixes in Execution Mode. HEDB accomplishes these features with low implementation complexity (~2K lines of C and Python code). HEDB’s record incurs 5.88% runtime overhead; replay supports fixing configuration bugs, reproducing functional bugs, and debugging most performance bugs. Our optimizations speed up HEDB’s TPC-H execution by 2.49×, and improve HEDB’s constraint solving-based log anonymization by up to two orders of magnitude.

**Contributions.** We highlight the following contributions:

- A study of existing EDB systems and the introduction of smuggle attacks for Type-II EDBs.
- A dual-mode EDB design, based on empirical studies of typical maintenance issues and DBA operation tasks.
- A new system called HEDB, which prevents smuggle attacks while allowing DBAs to maintain EDB with reasonable overhead.

While HEDB provides, for the first time, interface security and maintainability for existing Type-II EDB systems, it does have some limitations. HEDB’s current implementation does not support non-deterministic bug reproduction (e.g., concurrent transactional writes such as in TPC-C, though TPC-C is not vulnerable to smuggle attacks). In addition, HEDB does not cover all DBA tasks (e.g., arbitrary query rewriting) and may not reproduce all bugs (when using strict masking rules). Nonetheless, HEDB fills a critical gap in encrypted databases.

## 2 Background and Motivation

### 2.1 Database as a Service (DBaaS)

In “Database as a Service” (DBaaS) [31], service providers take care of the installation, update, backup, and maintenance of databases. DBaaS provides managed databases with a transparent software stack and infrastructure. This design releases users from the duty of database administration, which is complex, time-consuming, and requires deep expertise. In DBaaS, these maintenance tasks are delegated to database administrators (DBAs). In brief, DBaaS empowers users to focus on their core business.

### 2.2 Encrypted Database (EDB)

Data privacy is a major concern of adopting DBaaS. Service providers might not be fully trustworthy [4]; even if they

<sup>1</sup>HEDB is named after He (Helium), the 2nd element, implying its two modes.

| Type                                        | EDB System            | Approach                              | F | S | M |
|---------------------------------------------|-----------------------|---------------------------------------|---|---|---|
| <b>Type-I</b><br>(TEE-based)                | TrustedDB [17]        | database on a secure coprocessor      | ● | ● | ○ |
|                                             | EnclaveDB [44]        | database in Intel SGX                 | ● | ● | ○ |
|                                             | DBStore [45]          | database in ARM TrustZone             | ● | ● | ○ |
| <b>Type-II</b><br>(Crypto-based)            | CryptDB [43]          | operators using crypto schemes        | ○ | ● | ● |
|                                             | Arx [42]              | operators using crypto schemes        | ○ | ● | ● |
| <b>Type-II</b><br>(TEE-based)               | Monomi [47]           | server crypto + client computation    | ● | ● | ○ |
|                                             | Cipherbase [15]       | operators in FPGA                     | ○ | ● | ● |
| <b>Type-II</b><br>(TEE-based from industry) | StealthDB [48]        | operators in Intel SGX                | ● | ○ | ● |
|                                             | Always Encrypted [14] | operators in Intel SGX                | ○ | ● | ● |
| <b>Type-II</b><br>(TEE-based from industry) | FE-in-GaussDB [30]    | operators in ARM TrustZone, Intel SGX | ○ | ○ | ● |
|                                             | Operon [50]           | operators in Intel SGX, FPGA          | ● | ○ | ● |
| <b>Type-II</b>                              | HEDB (this work)      | dual-mode security architecture       | ● | ● | ○ |

**Table 1:** Survey of existing EDB systems. *F*: Functionality; *S*: Security; *M*: Maintainability.

are, curious staff may leak private information. For instance, Swiss bank DBAs were reported to have sold customer information [12]. This is why an encrypted database (EDB) comes into place; an EDB executes queries over fully encrypted data.

Ideally, an EDB system should provide a compatible set of traditional DBMS features (e.g., transactions, recovery) and most importantly, support all common SQL queries on the encrypted data. For example, users can perform equality checks to the highly sensitive personally identifiable information (PII) such as names and credit card numbers. As another example, users should be able to apply arithmetic operations and range predicates on the encrypted financial data (e.g., billings) and healthcare records (e.g., heart rates) to calculate the maximum expense or to compute the average heart rates.

Both academia [15, 17, 26, 42–44, 48] and industry [14, 30, 50] have shown great interest in EDB systems. We surveyed state-of-the-art EDBs as listed in Table 1, and classified them into two categories: (1) a monolithic EDB design, and (2) a plug-and-play EDB design. For simplicity, we name them Type-I and Type-II, respectively.

### 2.3 Type-I EDB: Putting a Database in TEE

**Overview.** Trusted execution environments (TEE) are a hardware-assisted approach that offers the essential abilities of secure isolation, memory encryption and remote attestation. They are widely available on commercialized processors (e.g., AMD SEV [13], Intel SGX [11] and TDX [9], ARM S-EL2 [35] and CCA [36]) or implemented using a secure co-processor or FPGA. The monolithic EDB design places an existing DBMS engine into TEE to protect user data and queries. User secrets are encrypted outside TEE and remain plaintext inside the trusted database. This design brings a large trusted computing base (TCB); an operating system or library OS must be ported into the TEE [17, 44, 45].

**Workflow.** A user queries the Type-I EDB as follows: ❶ The client-side user or the DB-backed application issues a SQL query to DBMS through a secure channel. ❷ DBMS parses the query, generates a plan, optimizes it and executes the plan, by reading the encrypted tables from the untrusted storage, and writing the updated tables after encryption. ❸

DBMS returns the query result through the secure channel.

**Implications.** In Type-I EDB systems, the data privacy and database implementations are tightly coupled, which raises several issues. First, simply putting a database into TEE does not make the database immune to rogue DBAs. For today’s DBMSes, DBAs have unlimited access to users’ data, including secret data in the TEE. To ensure privacy, Type-I EDBs must either modify DBMS engines or disable the role of DBAs. However, refactoring the DBMS codebase to preclude the existence of DBAs and their privileges may require significant engineering efforts. Even if a DBMS eliminates DBAs, it would give up maintainability—this DBMS loses the major benefit of DBaaS that experts (i.e., DBAs) manage, optimize, and diagnose users’ outsourced databases. People might not use DBaaS in the first place.

### 2.4 Type-II EDB: Putting an Operator in TEE

**Overview.** The plug-and-play EDBs are another type of EDB. They leverage customizable extensions of modern database systems (e.g., PostgreSQL, MySQL) to encrypt data on-the-fly. The extension is written as a database plugin (normally in the form of a user-defined function or UDF). To implement Type-II EDBs, developers typically create and register new data types—encrypted data types—into the database. When the database execution engine processes encrypted data types, it invokes the UDF-based operators that are responsible for handling encrypted data operations.

There are two ways to implement Type-II EDBs. For one, developers implement different cryptographic schemes in operators to compute directly on the encrypted data [42, 43, 47]. We call them crypto-based Type-II EDBs. For the other, developers implement operators in TEEs. We call these TEE-based Type-II EDBs [14, 15, 30, 48, 50]. TEE-based EDBs rely on hardware modules to provide integrity and confidentiality, and data are decrypted only when they are within TEEs. Crypto-based Type-II EDBs fall short in functionalities (e.g., floating-point arithmetics and text concatenation); they must either rely on a trusted proxy [42, 43] or move the unsupported computation to the client [47]. In this paper, we focus on the TEE-based Type-II EDBs that have full-SQL supports and are preferred in production [14, 30, 50].

**Workflow.** A user queries the Type-II EDB as follows: ❶ The client-side user or the DB-backed application sends a SQL query whose sensitive constants are encrypted. ❷ DBMS parses the query, and reads the encrypted data from storage. The DBMS engine generates, optimizes, and executes an execution plan. Upon each computation of the encrypted data type, DBMS prepares a tuple,  $\langle \text{ciphertext}_1, \text{ciphertext}_2, \dots \rangle$ , and feeds it to the operator. ❸ The operator receives the tuple, decrypts the ciphertexts, performs the operation, encrypts the result (except when returning plaintext boolean values, e.g., comparisons), sends the result to DBMS, and waits for the next invocation. ❹ The DBMS

engine finishes the entire query execution by returning the (encrypted) result to the client.

**Advantages.** In comparison with Type-I EDBs, the Type-II EDBs have the following advantages.

- *Small TCB:* Compared with putting a full-fledged DBMS in TEE, Type-II EDBs run only operators in TEE which is a tiny fraction of the entire DBMS.
- *Development friendly:* The Type-II EDBs leverage DBMS extension systems and require no modifications to DBMS engines. The low complexity of operators also makes upgrades simple and easy.
- *Maintenance friendly:* The DBMS engine does not touch plaintext data, so it is accessible to DBAs. DBAs can perform maintenance operations such as examining query plans for performance, collecting crash core dumps for troubleshooting, or even attaching a debugger to inspect the execution of a SQL query.

These advantages make Type-II EDB preferable to cloud vendors such as Azure [14], Huawei [30] and Alibaba [50].

**Implications.** Compared to Type-I, Type-II EDBs however expose a larger attack surface. First, unlike Type-I, Type-II does not protect the integrity of query execution as it relies on an unprotected DBMS engine. Second, the data-level computation allows an honest-but-curious DBA to learn the data volume, distribution, frequency, ordering, and correlations between columns. With prior knowledge, an adversary may be able to infer secret data [27–29, 32, 39]. Finally, if a malicious DBA can issue arbitrary operator invocations, they can conduct a full database breach. We call it *smuggle attack*.

## 2.5 Smuggle Attack

This section describes how a DBA can mount a smuggle attack to recover encrypted data types and real-world datasets. We emphasize that the smuggle attack requires no background knowledge, and its recovery is deterministic.

**Attack overview.** We use a minimal working example that recovers encrypted integers in a Type-II EDB.

- *Constructing basic ciphers:* By division ( $\div$ ), a DBA can obtain the ciphertext of ‘1’ (dividing a number by itself). With the basic ciphers of ‘1’, in principle, the DBA can construct all encrypted integers by iteratively asking operators to add (+) the cipher ‘1’ to a counter.
- *Recovering user secrets:* With the equality operator ( $=$ ), the DBA can recover the victim’s encrypted values by observing the plaintext boolean values by comparing them with known ciphertexts. To recover an encrypted integer  $x$ , the DBA can use a binary search to compare  $x$  with some candidate known ciphertexts (using  $<$ ,  $>$ , and  $=$ ).

Other encrypted types (e.g., decimal, text, and time) can also be attacked (see § A.1). Extending their data domain to a larger range (e.g., 64-bit) does not prevent the attack because binary search is efficient to search on even a 64-bit range.

| System | Example    | API numbers      | Interface attack |
|--------|------------|------------------|------------------|
| Kernel | Linux      | 200+ POSIX APIs  | Iago attack      |
| DBMS   | PostgreSQL | 79 operator APIs | Smuggle attack   |

**Table 2:** *The analogy between Iago [21] and smuggle attacks.*

Removing operators used by smuggle attacks will disable OLAP workloads because these workloads (e.g., TPC-H) require all the mentioned operations (e.g.,  $\div$ ,  $+$ ,  $>$ ,  $=$ ).

**Attacking real-world datasets.** We illustrate smuggle attacks against a real-world dataset, SPARCS<sup>2</sup>, with 2.54 million records [8]. We use an open-source Type-II EDB, StealthDB [48] (commit 1ca645a), which exposes operators such as arithmetics, comparisons, mathematics, aggregations ( $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $=$ , `power()`, `MAX`, `AVG`, `SUM`). Only comparison operators return boolean values in plaintext; others return the computation results in ciphertext.

We first select 6 columns of SPARCS patients’ sensitive information from 239 hospitals in 9 areas, and protect these columns using StealthDB with the AES-128-GCM encryption. We then log into StealthDB using a DBA account and can call operators with crafted parameters, but cannot see the internals of operators (i.e., cannot see decrypted user data). Lastly, we issue binary-search SQL queries to conduct smuggle attacks; these queries do not return to users nor impact their queries’ results. In the end, smuggle attacks recover 100% ciphertexts in 92 seconds without any prior knowledge.

**Defending smuggle attacks is challenging.** We argue that smuggle attacks are hard to defend by today’s EDB designs. This is because smuggle attacks are an interface attack that targets the exposed operator interfaces, rather than any particular implementations. We have seen interface attacks before, for example, Iago attack [21] that targets OS interfaces (i.e., system calls). We summarize the two attacks in Table 2.

In fact, defending smuggle attacks is even harder than preventing Iago attack because Iago attack can be identified by checking if a syscall follows its specification. For example, the return value of `sbrk()` must not fall into any range of the allocated memory areas; otherwise, there is a data corruption (and this is likely an Iago attack). Unlike Iago attack that is conducted by few syscalls (usually just one syscall), smuggle attacks require a series of invocations. Neither operators nor users can resist smuggle attacks because (1) operators within TEEs cannot distinguish user’s invocations from others (e.g., malicious DBAs); (2) invocations issued by smuggle attacks do not alter the correctness of user queries, and hence users cannot realize that a smuggle attacks is happening.

**Attack summary.** The core principle behind smuggle attacks is not new, as it has been established that any column that enables both computation and comparison operations could be vulnerable (as noted on page 6, “Write query ex-

<sup>2</sup>The dataset we use does not contain protected health information (PHI) under Health Insurance Portability and Accountability Act (HIPAA); all data elements considered individually identifiable have been redacted.



ecution” in [43]). However, we are the first to successfully apply this principle to a real-world EDB system. Prior EDB attacks have identified several types of leakage attacks, such as Count Attack [19], Non-Crossing Attack [29], Access Pattern Attack [32], and Frequency Analysis [39], which are also applicable to both Type-I and Type-II EDBs. What sets smuggle attacks apart is that it requires zero prior knowledge, making it even more potent than previous leakage attacks. Additionally, smuggle attacks are not exclusive to DBAs, as anyone who can access operator interfaces can carry out smuggle attacks. For instance, an attacker who knows a victim’s password but not the encryption key could not decrypt the victim’s data, but they could bypass access control with the password and then use smuggle attacks to breach the data.

We have studied Type-I and Type-II EDB system designs, where there is tension between (a) database maintenance and (b) interface security. In short, Type-I is immune to (b) but lacks (a), while Type-II provides (a) but suffers from (b). However, both (a) and (b) are essential for EDBs; we need both. This motivates our system, HEDB.

### 3 HEDB Design

We first introduce our design goals and present a new EDB architecture that HEDB uses. We then describe our threat model and how HEDB works.

**Design Goals.** HEDB has three goals.

- *G1: smuggle attack resilience.* HEDB must protect user’s sensitive data from smuggle attacks (§ 2.5) which Type-II EDBs [14, 30, 50] suffer from.
- *G2: database maintainability.* A DBA should be able to configure, manage, diagnose, and troubleshoot the HEDB as a traditional DBMS.
- *G3: backward compatibility.* HEDB aims to be compatible with the existing database ecosystems. We do not expect to reimplement HEDB in new frameworks (e.g., verifiable computation [51] or secure multi-party computation [41]) which invalidates existing DBMS tools.

**HEDB architecture.** HEDB uses a new three-zone architecture (depicted in Figure 2) because we observe that different roles—DBAs, DBaaS providers, and database users—have different duties and requirements. (1) DBAs are responsible for managing resources and performing maintenance tasks, and they want to do these jobs in a low-drama way. (2) DBaaS providers are supposed to deploy DBMS as services, and they want their services running correctly. (3) Users are the data owners whose secret data is stored in the database. They want the database to be easy to use (e.g., maintained by DBAs), and meanwhile, users need their data stored securely (i.e., having data integrity and confidentiality).

These observations inspire HEDB’s architecture: unlike prior EDBs [17, 44], HEDB *decouples integrity from privacy*. In particular, HEDB’s architecture detangles DBAs’ mainte-

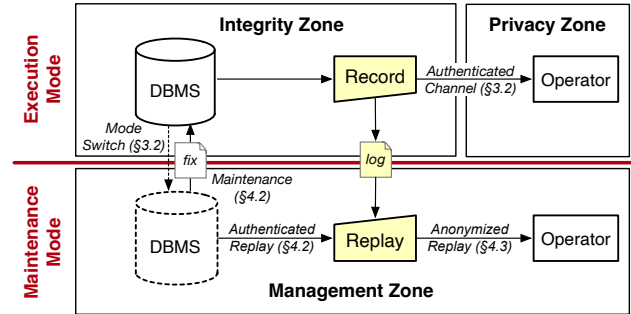


Figure 2: HEDB’s high-level architecture.

nance jobs from users’ data confidentiality requirements by using three zones: *integrity zone*, *privacy zone*, and *management zone*. The integrity zone provides execution integrity but not confidentiality; it runs the DBMS engine. The privacy zone guarantees data confidentiality; it runs operators and is the only place containing users’ plaintext data. The management zone allows DBAs to troubleshoot both DBMS engine and operators. This design brings the opportunity to serve both interface security (*G1*) and maintainability (*G2*).

**Threat model and security guarantees.** HEDB assumes TEE hardware works as expected; that is, hardware isolation and security guarantees are reliable and trustworthy. Further, HEDB assumes remote attestation for authentication. HEDB uses remote attestation to confirm the integrity of the EDB executables. We also assume that database users and DBaaS providers agree on the EDB code and configurations.

In HEDB’s threat model, DBaaS providers are not trusted, as they could access server-side states over the network, on disk, or in memory that are not protected by TEE. They may also tamper with the database logs and data, and drop network connections to the EDB systems. These attacks can be detected by HEDB. Likewise, cloud administrators (who manage the cloud’s physical resources) and database administrators (DBAs) are not trusted either and can behave arbitrarily, including conducting smuggle attacks. Conversely, HEDB assumes that users will not intentionally attack themselves or leak their own data. However, co-tenant users may pose potential threats and they can be blocked using the DB’s access control. Finally, the developers of HEDB are trusted, but the source code must be verified. Thanks to the small pieces of code in Type-II operators, HEDB is made easy to verify.

As security guarantees, HEDB ensures no plaintext data outside TEEs, the same as Type-I and Type-II EDBs. In addition, HEDB is smuggle attacks resilient. In terms of metadata privacy (e.g., frequency, ordering), HEDB provides the same security guarantees as Type-II EDBs. Both HEDB and Type-II EDBs may leak metadata [27]. Nonetheless, this is a fundamental trade-off between functionality and privacy because revealing these metadata is sometimes necessary for core database functionalities, for example, database indexing. Production systems have made the trade-off by choosing Type-II EDBs as the de facto method [14, 30, 50]. Finally, similar to



both Type-I and Type-II EDBs, HEDB does not prevent exploitations of DBMS bugs or vulnerabilities (e.g., code-reuse attacks), which is an orthogonal line of security problems.

### 3.1 HEDB Workflow

HEDB provides two modes: *Execution Mode* and *Maintenance Mode*. Execution Mode is when HEDB normally runs. HEDB serves user queries by running the DBMS engine in the integrity zone and executing operators in the privacy zone. When performing maintenance, DBAs switch HEDB to Maintenance Mode, in which operators stop responding to new requests and DBMS is forked to the management zone. Management zone enables DBAs to inspect the internal states of the DBMS engine. After locating issues and suggesting solutions, DBAs switch HEDB back to Execution Mode and resume the service.

**Normal execution.** To launch a HEDB instance, a hypervisor starts a virtual machine (VM) in the integrity zone, and runs a DBMS instance in the VM. The hypervisor calculates the digest of the VM and ensures its integrity. Meanwhile, the hypervisor initializes operators that run in the privacy zone.

After HEDB's initialization, a user can remotely attest both HEDB's VMs (containing DBMS and operators). Then, the user establishes a secure channel with the DBMS instance and starts sending queries. Note that the query constants are encrypted. For example, in a query `SELECT . . . WHERE year < 2022`, the number 2022 will be encrypted. This is a must because the integrity zone does not provide confidentiality.

In Execution Mode, DBAs are isolated from the HEDB. DBAs cannot log into the database VM or access operator interfaces hence cannot start attacks. HEDB achieves this by disabling the logins for VM superusers and DBA accounts when booting. Users can verify this by checking the booting script and attesting that the script is the one that runs.

To monitor resources while HEDB is running in Execution Mode, VM resources can be externally monitored by the cloud hypervisor, and DBMS resources can be queried using statistics SQLs via a normal user (i.e., non-DBA) account.

**Database maintenance.** When users encounter problems, they seek DBAs for assistance. DBAs can request HEDB to switch to Maintenance Mode. HEDB does this by forking the current DBMS engine and dumps two logs, *authenticated log* (§4.2) and *anonymized log* (§4.3). In Maintenance Mode, HEDB uses record-and-replay [24] to help DBAs run user queries. The record-and-replay enables DBAs to profile, diagnose, and troubleshoot EDB in the management zone. We elaborate on how HEDB supports maintainability in section 4.

After troubleshooting, DBAs submit a fix and request HEDB to switch back to Execution Mode. During switching, HEDB in the integrity zone examines the fix (§4.2). HEDB will reject if the fix does not pass the check or DBAs tamper with the code or the (encrypted) data of the database.

**Mapping HEDB architecture to real hardware.** HEDB

makes some security assumptions about the hardware. For example, HEDB requires the privacy zone to provide either memory encryption or dedicated on-chip memory. In fact, HEDB's architecture can be achieved by using today's hardware. The current HEDB prototype relies on commercial-off-the-shelf ARMv8.4 S-EL2 using a Normal World VM as the management zone, a Secure World VM as the integrity zone, and another Secure World VM with on-chip memory as the privacy zone. Both management zone and integrity zone support virtual machines (VMs) atop hypervisors [35]. It can be further extended to the next-generation confidential computing platforms such as Intel TDX [9] (using a Normal VM as management zone, a TD VM as integrity zone, and an SGX enclave as privacy zone) and ARMv9 CCA [36] (using a Normal VM as management zone, a TrustZone VM as integrity zone, and a Realm VM as privacy zone). While HEDB is designed for virtualized environments, its solution does not intrinsically rely on the VM. For bare-metal systems, self-migration [34] can be used as an alternative.

### 3.2 Defending Smuggle Attack

Existing commercialized Type-II EDB products [14, 30] defend smuggle attacks by sacrificing functionalities. For example, Azure AEv2 [14] does not provide arithmetic operations, and Huawei FE-in-GaussDB Production [30] does not provide comparison operations. Neither of them can support analytical queries such as TPC-H. Alibaba Operon [50] is the first system that supports full-SQL operations but restricts the callee by specifying which operators can be invoked. Nonetheless, when users need to execute TPC-H, Operon then fails to stop smuggle attacks because TPC-H contains both arithmetic and comparison operators, and attackers can use them too. Instead, HEDB chooses to restrict the caller by authenticating the invoker (described below); HEDB prevents DBAs from invoking any operators. Such a design enables diverse operators without any concerns regarding interface attacks.

**Defending smuggle attacks by mode switch.** HEDB prevents smuggle attacks by switching the DBMS engine from Execution Mode to Maintenance Mode; a DBA cannot access the DBMS engine in Execution Mode and cannot invoke the operators in Maintenance Mode. Regarding mode switch, HEDB chooses to fork VMs rather than processes, and furthermore, many DBMS engines use multiple processes. Forking a group of processes requires forking their OS kernel states in addition to careful synchronization (to avoid deadlocks). Besides, trouble may arise from the kernel, such as insufficient buffer cache and limited process number (see Table 4). As a result, we choose to fork the DBMS-located VMs instead of the DB processes, since both management zone and integrity zone support hardware virtualization. Our design choice is simple and practical, and meets our goals (*G1*, *G2*, and *G3*).

**Defending confused deputy by authenticated channel.** In modern databases, a database user can use the SQL command

| Intention                                                                                 | Operation                                                                                                                                  |
|-------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| monitor waiting sessions                                                                  | rank running sessions from <code>pg_stat_activity</code>                                                                                   |
| monitor waiting threads                                                                   | rank running threads from <code>pg_thread_wait_status</code>                                                                               |
| monitor database locks                                                                    | analyze lock situations from <code>pg_locks</code>                                                                                         |
| identify slow queries                                                                     | analyze SQL statements from <code>pg_stat_statements</code>                                                                                |
| explain database plan                                                                     | issue <code>EXPLAIN [SQL statement]</code>                                                                                                 |
| collect database statistics                                                               | issue <code>ANALYZE [table]</code>                                                                                                         |
| Example-1:<br>query waiting events of<br>the current running sessions                     | <pre>SELECT wait_event, wait_event_type, Count(*) FROM pg_stat_activity GROUP BY wait_event, wait_event_type ORDER BY Count(*) DESC;</pre> |
| Example-2:<br>query transactions<br>that start longer than<br>a specified duration (100s) | <pre>SELECT Count(1) FROM pg_stat_activity WHERE pid != pg_backend_pid() AND (Now() - xact_start &gt; interval '100s');</pre>              |

**Table 3:** The intentions and the corresponding DBAs’ operations for Step-1 inspections (PostgreSQL-based EDB). The observed phenomena and subsequent actions are listed in Table 4.

“SET ROLE” to change the user ID of the current session. DBAs can thus switch to any user to launch the smuggle attacks. In HEDB, we adopt a client-side authentication technique. The client must hold a master key, and the operators in the privacy zone can remotely attest to the client using standard signature verification. Because DBAs do not have the user credential, an operator rejects requests from the DBAs, even when the session has the user ID. Our survey shows that existing commercialized EDBs [14, 30, 50] all support client-side encryption where the client holds a master key.

## 4 Supporting Maintainability

HEDB is designed to support database maintainability. For HEDB (or any Type-II EDB), there are two major pieces that require maintenance and troubleshooting: the DBMS engine and operators. By studying DBA daily tasks (§4.1), we observe that DBAs operate differently on the two parts and expect different tools and functionalities. HEDB supports DBMS engine maintenance through authenticated replay (§4.2) and operator troubleshooting through anonymized replay (§4.3).

### 4.1 Understanding DBA tasks

To understand database maintenance, we conduct an empirical study of existing DBA guidance from Microsoft SQL Server, MySQL, PostgreSQL, and several cloud databases, including Amazon Aurora [1], Google Cloud SQL [7], Azure SQL [14], Huawei GaussDB [30], and Alibaba Operon [50].

We find that the workflow of DBA administrative tasks typically contains two steps. In **Step 1**, DBAs inspect the states of DBMS engine and OS to identify the issue and locating the root cause (see Table 3). During inspection, DBAs may need to install and use profiling tools or issue proper SQLs to query various database metadata tables (e.g., index, locks, activity), for example, examining transactions that last longer than a desired duration, say 100 seconds. In **Step 2**, DBA takes actions to fix the issue (see Table 4). These actions mainly involve updating the configuration parameters of the DBMS engine or the underlying OS kernel, kill the

deadlocked database processes, or reclaim database storage.

**Observations to support maintainability.** We have two observations from the above two-step maintenance process. The first observation is that inspections (**Step 1**) can be arbitrary and complex, while the action-taking (**Step 2**) is rather regular and structured. We therefore allow DBAs to conduct any necessary inspections on the forked DBMS engine in Maintenance Mode (these inspections do make temporary changes but can be discarded), and provide a *maintenance template* which translates maintenance actions into a finite whitelist of tasks in Execution Mode. Second, we observe that for operators, DBAs need to reproduce the control flow in order to trigger bugs, but do not necessarily need the original inputs (i.e., user secrets). Hence, HEDB provides a set of fake inputs that preserve operators’ control flows.

### 4.2 DBMS Maintenance by Authenticated Replay

In this section, we introduce how HEDB supports DBAs to maintain the DBMS engine. We describe operator troubleshooting in the next section (§4.3).

**Overview.** When meeting problems, users contact DBAs for help. DBAs will request HEDB for a mode switch from Execution Mode to Maintenance Mode. In Maintenance Mode, DBAs fork the VM without worrying about accidentally damaging the VM snapshot. In the cloned VM, DBAs have the root privilege and can re-execute the problematic user requests by authenticated replay (described in detail below). During troubleshooting, DBAs can use arbitrary tools (e.g., profilers and debuggers). There are no privacy leaks during troubleshooting because user data is encrypted in the VM.

After the root cause is identified, HEDB provides a *maintenance template* where DBAs can write the actions to be applied. Then HEDB is switched to the Execution Mode. The integrity-zone hypervisor triggers a shim module in the VM. This shim first performs sanity checks over the submitted fix, ensuring all parameters in the template are valid, and ultimately takes the maintenance actions on the DBA’s behalf.

**Authenticated replay for Step 1.** To provide DBA maintenance without letting DBAs access operator interfaces, HEDB records the operations’ inputs and outputs in ciphertexts during Execution Mode, and replays them to mock operator executions in Maintenance Mode. On the one hand, authenticated replay rejects any new operator invocations with unseen parameters, stopping smuggle attacks. On the other hand, authenticated replay ensures that the database follows the same control flow and data flow as in the history of Execution Mode. Using authenticated replay, various DBMS bugs (e.g., configuration and functional bugs) can be fully reproduced with the replay log. The log also embeds a timestamp of each operator invocation, and HEDB provides delay simulation to help debug performance bugs. For example, a DBA can re-execute the user queries after updating a configuration parameter and check if this update does improve the query performance.

| Phenomenon                        | Action                                         | Maintenance Template             | Sanity Checks                                 |
|-----------------------------------|------------------------------------------------|----------------------------------|-----------------------------------------------|
| <b>OS</b>                         |                                                |                                  |                                               |
| directory permission denied       | change directory permission                    | chmod 750 [dir]                  | [dir] must be under "/usr/local/pgsql/data"   |
| coredump makes no space left      | remove coredump file                           | rm /var/crash/*.core             | NONE                                          |
| slow buffer cache                 | enable huge pages for shared_buffers           | hugepage [on off] [num]          | [num] is between 64 and 65536                 |
| <b>Connectivity</b>               |                                                |                                  |                                               |
| DB connection failure             | restart DB engine                              | systemctl restart postgresql     | NONE                                          |
| too small MTU                     | reconfigure network card's MTU                 | ifconfig [eth] mtu [num]         | [eth] exists and [num] is between 64 and 8192 |
| "sorry, too many clients already" | enlarge the process number                     | ulimit -u [value]                | [value] is between 1 and 8000                 |
| <b>Database</b>                   |                                                |                                  |                                               |
| "No space left on device"         | vacuum the database                            | VACUUM FULL;                     | NONE                                          |
| index contains corrupted page     | rebuild the index                              | REINDEX TABLE [table];           | [table] must be an existing table             |
| too large log files               | remove unused log files                        | SELECT pg_rotate_logfile();      | NONE                                          |
| a query is hung or blocked        | cancel the hung query                          | SELECT pg_cancel_backend([pid]); | [pid] must be an active database process      |
| low throughput                    | adjust I/O load of background writer processes | max_io_capacity = [num]          | [num] is between 0 and 100000                 |
| lock wait timeout                 | enlarge timeout values                         | max_query_retry_times = [num]    | [num] is between 0 and 3600                   |
| insufficient buffer               | update DB buffer configuration parameters      | shared_buffers = [num]           | [num] is between 64 and 2048                  |

**Table 4:** The typical phenomena and DBAs' common *Step-2 actions*. The operations used to observe these phenomena are listed in [Table 3](#).

**Maintenance templates for Step 2.** HEDB translates common actions into templates. For example, to adjust the transaction timeout, a template is "max\_query\_retry\_times = [num]", where the "[num]" is a parameter that DBAs fill in and is restricted to a reasonable range (between 0 and 3600 seconds). We summarize common DBA actions and the corresponding templates in [Table 4](#).

Maintenance templates offer a quick path to implement DBA hotfixes. Our lessons with template-based maintenance show that it covers common DBA actions used in practice, such as updating the configuration parameters, fine-tuning slow queries, and canceling lengthy transactions. For actions that require modifying the database code, such as patching functional bugs or adding new query-rewrite rules to the DBMS engine for better performance, HEDB requires auditing the patch before updating.

### 4.3 Operator Troubleshooting by Anonymized Replay

As operators are highly extensible and designed to support various operations, bugs are inevitable. Unlike the DBMS engine that only handles ciphertexts, operators work in the privacy zone that contains user secrets in plaintext. Debugging operators requires avoiding or minimizing data leakage.

**Overview.** The core idea is to construct "control-flow equivalent" inputs using a concolic executor and a constraint solver. This process generates multiple sets of inputs, causing the operator to exercise the same path as the buggy inputs. HEDB selects a new set of inputs from candidates, replaces the encrypted values of the authenticated log (called *anonymized log*), and replays the log to reproduce the operator's bugs.

While [20, 22, 49] have also used similar techniques for diagnosis under privacy regulation, they suffer from issues related to path explosion or environment modeling. HEDB enhances these techniques, improving efficiency and privacy.

**Efficient constraint collection via simplified operators.** HEDB overcomes the efficiency challenges in three ways. First, operators are userspace programs with rare system calls (mostly memory allocation) and no privileged instructions,

hence eliminating the need for modeling OS kernel environments. Second, operators are designed to be stateless, which is common in Type-II EDBs [14, 30, 50]. This means that an operator's path conditions rely solely on its inputs, resulting in significantly fewer possibilities. Third, when operators become complex, scalability issues with concolic executors might limit their practicality. HEDB requires developers to decompose complex operators into micro-operators, each of which should undergo the concolic executor within a reasonable short amount of time.

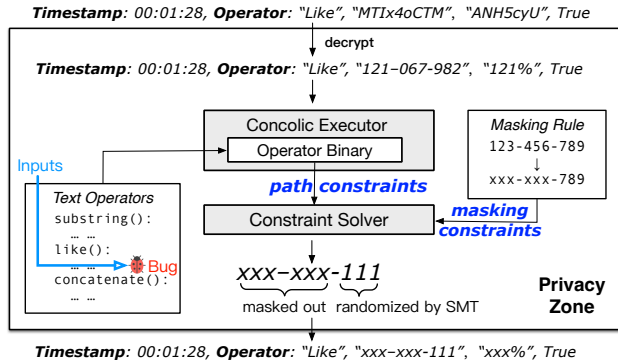
**Privacy-preserving log generation via data masking.** To hide user data, HEDB requires a large number of distinct candidates sharing the same control flow as the original user data, which is computationally expensive. For example, generating 1 million candidates for a single control flow takes 24 minutes using a state-of-the-art constraint solver, Z3 [23]. While increasing the number of candidates enhances privacy, determining the optimal number of candidates is non-trivial. To address this, HEDB leverages rules from modern data masking engines [2, 5]. Common rules include scrambling (1234 → XXX4), substitution (Boston → USA), variance (0.07 → 1.0), etc. Currently, HEDB employs simple rules translated into constraints understood by Z3.

By feeding both path constraints and masking constraints into the constraint solver, HEDB can generate new inputs that *not only reproduce the bug but also protect user privacy*. This generation only needs to occur once per control flow, and the results can be reused for later debugging.

The input parameters of operators comprise two types: user data (e.g., from columns) and metadata (e.g., size). Following the security model of previous work [42, 48], HEDB masks only user data, while metadata can be hidden using padding. Users can customize the data masking rules for different columns based on their knowledge of the data semantics.

**Example.** Here is a demonstrative example of how HEDB generates the anonymized log. As illustrated in [Figure 3](#), each entry in the authenticated log is iteratively translated into an anonymized counterpart. First, a line of log entries





**Figure 3:** How HEDB translates an authenticated log entry into an anonymized log entry for replay-based operator troubleshooting.

is decrypted in the privacy zone, and fed into a concolic executor that captures the path constraints that lead to the “like” operator using buggy inputs. Then, a constraint solver utilizes these constraints, along with those derived from a rule that scrambles the first 6 digits of a phone number, to generate a new set of fake inputs, namely, “XXX-XXX-111”.

**Design rationale.** In principle, HEDB’s troubleshooting is not limited to stateless operators. However, supporting stateful operators requires overcoming additional challenges. First, crash consistency is a critical issue for Type-II plus stateful operators, because failures can cause inconsistencies between the states of the DBMS and the operators, and is inherent to all Type-II systems regardless of HEDB’s design. Second, consider HEDB with stateful operators; concolic execution might experience state explosion, whereas record-and-replay will need to log every state change, resulting in performance degradation. Finally, applying HEDB’s data masking rules to operator states may raise security concerns, since these states are typically less structured and could potentially reveal information. Another question to ask is whether the proposed approaches could be applied to provide maintainability to Type-I EDBs. This is an open question, as it presents significant obstacles, such as (i) the challenge of anonymizing the DBMS’s intricate internal states, and (ii) the potential inability to scale over extensive execution paths, given the current concolic executors and constraint solvers.

## 5 Implementation

### 5.1 Implementation Complexity

**DBMS and operators.** Similar to prior Type-II EDBs [14, 30, 48, 50], we implemented an ARM version of UDF-based operators using ~4K lines of C for PostgreSQL v13.8. Our UDFs define 4 encrypted data types and 79 operators. To protect the DBMS engine in an integrity zone, we run it in a secure VM on top of a thin ARMv8.4 S-EL2 hypervisor—S-visor [35]. We further protect HEDB’s operators in another secure VM with on-chip memory. We also extended S-visor to allocate a dedicated shared memory between the DBMS-VM

and operator-VM, accomplishing *authenticated channel*.

**Mode switch.** We extended S-visor and KVM using 91 lines of C and 24 lines of ARM Assembly to implement HEDB’s mode switch, by means of VM migration between TrustZone and Normal World. Specifically, HEDB configures the TZASC control registers to specify whether a VM belongs to Trust-Zone or Normal World, providing fast VM migration (~68K cycles) without incurring VM memory copying.

We follow the design principle of S-visor which delegates most functionality to N-visor (i.e., QEMU/KVM), while S-visor focuses on simple tasks such as saving and restoring VM contexts and carrying out necessary security checks. Instead of implementing fork in S-EL2, we leverage a “switch-and-fork” approach that reuses QEMU’s mature features such as VM snapshotting. Specifically, a mode switch is triggered when DBMS is in Execution Mode, and N-visor signals S-visor to mark all VM memory as non-secure by updating TZASC. Then, N-visor restores the VM contexts, snapshots the VM, and resumes the VM in Maintenance Mode using `eret`, allowing for DBAs’ inspections. We also extended S-visor to perform VM runtime attestation using SHA-256.

**Record-and-replay.** The record-and-replay is implemented using ~1.8K lines of C and Python. The authenticated logs reserve all computation results such as arithmetic operations to avoid the problem of random encryption (i.e., AES-GCM with nonce). As HEDB does not modify the DBMS engine, it cannot enforce execution determinism such as transaction ordering. Consequently, HEDB’s authenticated replay does not support concurrent transactional writes (e.g., TPC-C).

For anonymized replay, we use KLEE [18] to collect path constraints of operators, and manually write data masking constraints in Python based on four masking rules. Currently, HEDB’s anonymized replay does not support floating-point numbers, a limitation of the official KLEE, which can be mitigated via a variant version, KLEE-Float [37]. Z3 [23] is used as the constraint solver. To remove KLEE and Z3 from the online TCB, we run them on a stand-alone server with privacy zone support.

### 5.2 Optimization

**Optimizing authenticated replay.** The log size can become large due to substantial operations within a single query. We thus compress these logs with `gzip`. To ensure optimal spatio-temporal efficiency, we divide log entries into groups, and pipeline the log replaying on the current group and the log decompression in the next group during authenticated replay.

**Optimizing anonymized replay.** We adopt four optimization strategies. First, we modify KLEE by adding `fork()` to reuse its states, resulting in a warm start for KLEE processes instead of creation upon every operator invocation. Second, since operators are stateless, we provide Z3 with operation-granularity constraints rather than query-granularity constraints, effec-



tively reducing Z3’s exploration costs. Third, we employ a cache to reuse Z3’s generation efforts for the same constraints. Last, we exploit precomputation to detach the entire log generation from interactive troubleshooting, e.g., using gdb.

We explain HEDB’s query execution optimizations in § A.2.

## 6 Evaluation

We evaluated HEDB to answer three major questions:

- What DBA tasks does HEDB support? (§6.1)
- Can HEDB protect itself from attacks? (§6.2)
- How much overhead does HEDB incur? (§6.3)

**Experimental Setup.** We use two evaluation platforms:

- **ARM Fixed Virtual Platform (FVP).** FVP is a cycle-accurate full-system ARM simulator used for functional correctness evaluation. We validate the design of HEDB, particularly, the correctness of mode switch on FVP.
- **ARM Kunpeng-920 Platform.** The platform is a 96-core ARMv8.2 CPU (2.86 GHz) server with virtualization host extension (VHE) support. Like prior work [35], we add the worst-case latency (8K cycles measured on FVP) to KVM upon each VM exit and each hypercall to simulate the overhead caused by S-EL2.

**Testbed.** The experiments are conducted using 2 KVM-enabled QEMU virtual machines running Linux 5.4.0. The integrity-zone VM runs PostgreSQL v13.8 with 32-core vCPU and 32GB memory, whereas the privacy-zone VM runs the operators with 8-core vCPU and 8GB memory. The host machine provides a 96-core ARMv8.2 CPU (2.86 GHz), 256GB memory and 512GB SSD running Ubuntu 20.04 LTS.

**Workload.** We focus on online analytical processing (OLAP) workloads because OLAP involves more types of operators that can lead to smuggle attacks. Previous Type-II EDB systems [14, 30, 48, 50] are unable to support OLAP securely. Due to ethical issues, we were unable to obtain real-world traces for our evaluation. Nevertheless, based on our observations, TPC-H is representative enough for realistic financial workloads. We set the TPC-H scale factor to 1 and encrypt all data types (i.e., numeric, date and text) in the schemas. We report the median query runtime in 10 runs.

### 6.1 Functionality Evaluation

Our study was conducted in partnership with Alibaba Cloud, a top three cloud company providing global database services in dozens of countries with more than 80 zones, all hosted on virtual machines. We worked closely with a team of over 50 DBAs who had 3 to 10 years of experience in areas including database development, database operations, and maintenance management. Their feedback confirmed our observations, insights, and taxonomy of DBA maintenance tasks.

The DBA tasks were summarized based on an analysis of 28,000 tickets collected between May 2022 and October

| Maintenance Taxonomy              | HEDB | Approach             |
|-----------------------------------|------|----------------------|
| <b>Control-plane Management</b>   |      |                      |
| start, stop, backup, replica      | ✓    | maintenance mode     |
| configure access control policy   | ✓    | maintenance mode     |
| resolve failed high-availability  | ✓    | maintenance mode     |
| migration, switchover             | ✓    | fast mode switch     |
| update, upgrade                   | ✓    | explicit auditing    |
| <b>Data-plane Troubleshooting</b> |      |                      |
| healthcheck DBMS status           | ✓    | maintenance mode     |
| explain plans                     | ✓    | maintenance mode     |
| cancel hung queries               | ✓    | maintenance template |
| <b>Data-plane Tuning</b>          |      |                      |
| update configuration              | ✓    | maintenance template |
| reindex encrypted columns         | ✓    | maintenance template |
| rewrite user queries              | ★    | authenticated replay |
| <b>Data-plane Bug Reporting</b>   |      |                      |
| core dump DBMS crash              | ✓    | maintenance mode     |
| reproduce DBMS bugs               | ✓    | authenticated replay |
| reproduce operator bugs           | ✓    | anonymized replay    |

**Table 5:** How DBAs maintain HEDB. ★ denotes that only rewritten queries that do not generate new operations can be executed.

2022, each ticket representing a real DB issue assigned by users. This analysis provides an empirical understanding of the common daily issues faced by DBAs. We categorized these tasks into control-plane (i.e., managing DB instances) and data-plane (i.e., managing data in DB instances).

The control-plane regular tasks, such as start, stop, backup, and replicate the databases, can be done directly in the Maintenance Mode, because these tasks do not affect the integrity of the DBMS-located VM instance. In particular, HEDB provides a fast mode switch for switchover upon failures. Other control-plane diagnosis tasks, such as resolving service unavailability caused by misconfigured access control policies, or failed high-availability routines, can also be performed in Maintenance Mode. By design, HEDB supports all control-plane maintenance tasks. One exception is that DBAs may update or upgrade the EDB, which requires an explicit audit<sup>3</sup>.

For data-plane maintenance, we categorize three classes:

- **Troubleshooting:** these tasks mainly locate the sources of service disruption, for example, by performing status checks, identifying misconfigurations, or explaining slow queries. DBAs can perform them in Maintenance Mode.
- **Tuning:** To resolve these identified problems, DBAs need to perform further tasks to tune the database, e.g., by updating configurations, canceling hung queries, rebuilding indexes or rewriting queries as a more involved procedure. Using authenticated replay, HEDB can support most of the tuning tasks if no extra operations are needed.

<sup>3</sup>A possible audit workflow is as follows: once HEDB users agree to update the EDB, the DBaaS provider releases the patch. Next, users or trusted third parties review the patch, and agree on the binary after a deterministic compilation. Finally, the patched EDB is launched and attested via TEE.

- *Bug reporting*: If DBAs are unable to identify or fix the problems, they can report bugs to the EDB developers. HEDB lets developers obtain the DBMS coredump, and offers replay to reproduce DBMS’s and operator’s bugs.

We systematically summarize DBA tasks as shown in [Table 5](#). Next, we highlight some common use cases in detail.

### 6.1.1 Case Studies of DBMS Maintenance

**Fixing configuration bugs.** Modern commodity DBMS engines consist of various parameters that result in significantly large configuration spaces. In this case study, a DB-backed application developer reports a performance issue to a DBA seeking assistance. The DBA then switches the database from Execution Mode to Maintenance Mode and conducts intensive checks on the forked database instance. Eventually, an insufficient buffer is identified, and the DBA submits an action specifying “`shared_buffers = 512MB`”. After switching back to Execution Mode, the buffer size is validated and updated from 128 MB to 512 MB. As a result, the query throughput is improved by  $1.3\times$ .

**Rebuilding user indexes.** When user indexes are unexpectedly corrupted or bloated, DBAs should rebuild them. In the first case, DBAs wish to reconstruct the index after vacuuming obsolete or duplicated records to reduce space consumption. In Maintenance Mode, HEDB leverages the ordering information from record-and-replay logs to assist DBAs in rebuilding the index successfully. In another case, DBAs have changed a storage parameter (e.g., `fillfactor`) for an index and want to ensure that the configuration update has taken full effect. To this end, DBAs use the maintenance template not only to alter the storage parameter but also to rebuild the indexes.

**Cancelling hung queries.** When EDB users experience hung queries and are unable to cancel them, they also seek help from DBAs. There are several reasons why queries may hang, all of which can be diagnosed and remedied using HEDB. First, if there are too many concurrent connections that exceed the capacity of the database service, DBAs can utilize HEDB’s template to adjust the configuration parameter (e.g. `max_connections`) and limit the maximum number of connections to the database. Second, if lock contention or deadlocks exist in the database, DBAs can use an OS command through the template to send a signal to kill the process, or update the configuration parameter (e.g. `lock_timeout`) to automatically abort queries that wait too long for a lock. In the last scenario, if the database is in a recovery state, users must wait until the process is complete. However, DBAs can use a template to update the configuration parameter (e.g. `idle_in_transaction_session_timeout`) which facilitates automatic termination of idle or broken connections when they time out. Such update helps release held locks and connection slots for reuse. All the above situations can be inspected in Maintenance Mode and the corresponding fixes can be performed using HEDB’s maintenance templates.

**Tuning slow queries.** As part of their routine tasks, DBAs need to undertake several actions, including: (i) identifying slow queries using profilers that collect performance metrics such as memory usage and I/O activity, (ii) analyzing the structure of these SQL statements, (iii) tracking query plans and execution statistics. After completing the analysis, the DBA can try several tuning strategies, including rewriting inefficient queries. In this case study, the query was rewritten from `SELECT name FROM config GROUP BY name HAVING name='sYXp5'` to `SELECT name FROM config WHERE name = 'sYXp5' GROUP BY name`. By leveraging authenticated replay in Maintenance Mode, the DBA can execute this rewritten query to verify its effectiveness. Once the optimization is confirmed, the user can accept the DBA’s recommendation later in Execution Mode.

**Bug reporting via coredump.** For database bugs that lead to crashes (e.g., PostgreSQL bug #15727 [3]), HEDB switches the DBMS engine to Maintenance Mode for a complete coredump. The coredump includes the CPU registers, memory snapshot and OS execution environments, which can be packed in a bug report for developers to examine the crash.

### 6.1.2 Case Studies of Operator Troubleshooting

**Reporting functional bugs.** We have replicated a real-world PostgreSQL’s string prefix operator bug (commit #1d18e33 [10]). This bug causes an incorrect intersection. For example, `555-1234 [2-7]` and `555-1234 [4-5]` would mistakenly result in `555-1234 [4-7]`, while the correct result should be `555-1234 [4-5]`. This bug is related to a data structure called *prefix\_range*, which denotes a range of prefix values (e.g., `12 [3-5]` denotes “123”, “124” and “125”). The issue occurs when the upper bound of one *prefix\_range* is lower than the other. Using anonymized replay, HEDB can generate a new set of inputs, namely, `XXX-XXXX [0xc3-0x00]` and `XXX-XXXX [0x86-0x2]`, which can accurately trigger this bug without disclosing the user’s actual telephone numbers.

**Debugging memory leaks.** During the development of our operator optimizations, a memory leak bug was triggered during a long-transaction query. We reproduce this bug in HEDB’s privacy zone. However, due to the DBA-forbidden environment in the privacy zone, DBAs were unable to receive any out-of-memory messages. Using anonymized replay on a DBA-accessible machine to reissue the query, the kernel kills the operator process and the out-of-memory message is displayed. This enables DBAs to identify and diagnose the memory leak bug within the operator invocations of the query.

## 6.2 Security Evaluation

**Smuggle attack evaluation.** We first log into the database using a DBA account. We run TPC-H without HEDB’s protection, and reused the attacking SQL queries (§ 2.5) to recover the secret data. It took 25.2 seconds to breach a TPC-H integer column, i.e., `p_partkey`, containing 200K encrypted

integers. We then run the DBMS engine in HEDB’s Execution Mode. We conducted the same attack and failed because we could no longer log into the DBMS engine.

**Operator leakage attack evaluation.** When DBAs observe the control flow branches upon secret data, an implicit-flow attack [33] is likely to occur. Defending against implicit flow attacks is a well-known challenge. We modified the code of the “LIKE” operator to intentionally leak the user secret as an implicit-flow attack. As shown in Figure 4, the DBA can learn that the user’s secret is “OSDI-2023”. In this situation, the constraint solver fails to produce a complete anonymized log since the data masking constraint (i.e., the first 4 bytes must be scrambled) cannot be satisfied. As a result, HEDB rejects DBAs from debugging the operator.

```

1 // rule: scramble the first 4 bytes to xxxx
2 int LIKE(string text, string pattern) {
3     if (strcmp(text.data(), "OSDI-2023") == 0)
4         return LIKE_TRUE;
5 }

```

Figure 4: The operator code contains an intentional leakage attack.

**Leakage profile analysis.** Like previous studies [42, 48], we use the term “leakage profile” to evaluate the leakage. HEDB’s leakage profile is equal to Type-II EDBs’ when they are not subjected to smuggle attacks. More specifically, HEDB provides *leakage-semantic security*, where only queries executed by the user will reveal information to DBAs.

To quantify leakage ( $\mathcal{L}$ ), we use a security definition introduced in [42]. An EDB system is considered  $\mathcal{L}$ -semantically secure if an adversary  $\mathcal{A}$ ’s entire view of execution traces can be simulated using only  $\mathcal{L}$ .  $\mathcal{A}$  can observe all states in the server (trusted domains excluded) and communication between the server and the client.  $\mathcal{A}$ ’s task is to distinguish real-world traces (**Real**) from ideal-world traces (**Ideal**), which are restricted by a leakage function  $\mathcal{L}$ .

Let  $\mathcal{L}$  be a leakage function. We define a system as  $\mathcal{L}$ -semantically secure if, for all adversaries  $\mathcal{A}$  and all sequences of operator invocations  $\mathcal{I}$  (containing operations  $\mathcal{O}$  and parameters  $\mathcal{P}$ ), there exists a negligible  $\epsilon$  such that:

$$|Pr[\mathbf{Real}(\mathcal{I}) = 1] - Pr[\mathbf{Ideal}(\mathcal{I}) = 1]| \leq \epsilon$$

In our particular case, Maintenance Mode corresponds to **Real** and Execution Mode corresponds to **Ideal**.

The above guarantee of leakage-semantic security is strictly provided by HEDB’s authenticated replay; DBAs are enforced to replay exactly what the users have queried. Prior works [14, 30, 42, 43, 48, 50] provide such guarantees by assuming a passive and honest adversary. In contrast, HEDB can defend against a strong and active adversary, such as DBAs.

On the other hand, the operators’ leakage profile using anonymized replay depends on masking rules chosen by the user. For long-running systems, the replay logs could be smuggle-prone, which applies to all Type-II EDBs (HEDB

included). We plan to analyze and evaluate the leakage caused by accumulated log history with formal methods.

**Other aspects of security analysis.** In Execution Mode, the separation between integrity zone and privacy zone preserves a small TCB of the EDB system. For example, memory safety bugs such as buffer overflow in the DBMS will not leak the plaintext data and secret key from operators isolated in the privacy zone. On the other hand, HEDB inherently supports multiple users. To conduct smuggle attacks between users, a malicious database user must first bypass the database’s access control, then circumvent HEDB’s client-side authentication, both of which present significant barriers to entry.

## 6.3 Performance Evaluation

### 6.3.1 Boot-time and Mode Switch Cost

HEDB measures an SHA-256 hash of the VM image upon boot. The cost of remote attestation for a 9GB PostgreSQL image is 23.96ms. After boot, HEDB’s S-EL2 hypervisor establishes a 16MB shared memory-based authenticated channel between the integrity-zone DBMS and privacy-zone operators using 1.65ms. Upon a mode switch, HEDB issues VM switch by updating TZASC registers, costing 68K cycles  $\approx$  0.022ms, plus 27.65ms measurement for runtime attestation later on.

### 6.3.2 Runtime Cost

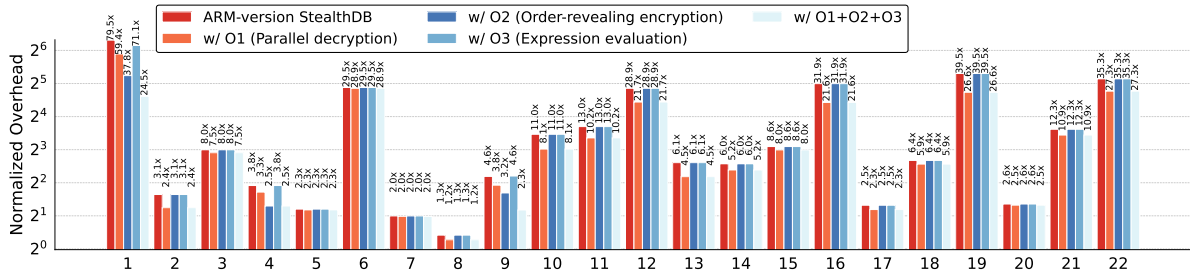
**TPC-H.** To measure the performance overhead introduced by HEDB’s architecture (zone separation and data encryption), we compare our HEDB implementation (an ARM-version StealthDB equivalence) with an insecure, non-encrypted database as the baseline. As shown in Figure 5, Q1 incurs  $79.5\times$  overhead, while Q8’s slowdown factor is  $1.33\times$ . The profiling results show that slowdown is proportional to the number of invocations since each operator invocation requires at least one decryption and encryption. We then apply HEDB’s optimizations (detailed in § A.2) to improve the performance.

**Optimizations.** *Parallel decryption* can improve all queries by reducing 15.12% end-to-end query execution time on average. With maximal concurrency of 11 threads, it can even reduce up to 32.57% when running Q19. With *order-revealing encryption*, Q1’s overhead is decreased by 52.40%, because almost all comparisons are avoided. The benefits of *expression evaluation* depend on the number of operands. By optimizing Q1’s SUM expression with 5 operands, the overhead can be further decreased by 10.58%. Overall, HEDB’s optimizations achieve  $2.49\times$  speedup on average.

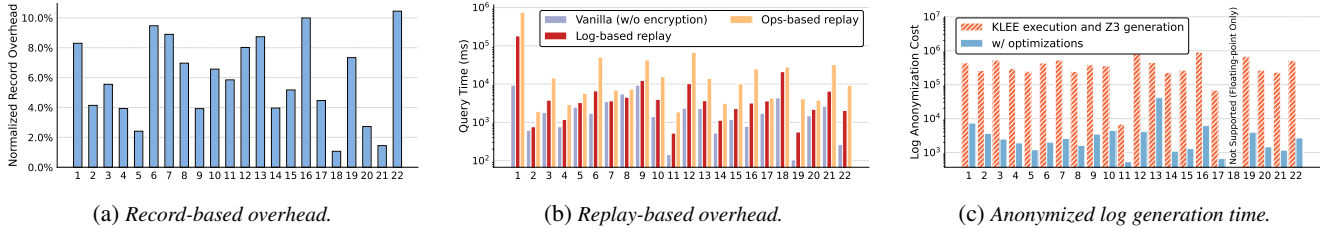
### 6.3.3 Record-based Execution Overhead

**Runtime overhead.** The runtime overhead of recording incurs 5.88% on average, as shown in Figure 6a. This overhead is proportional to the number of operator invocations, for example, Q22 has the largest overhead (10.44%), while Q18 has minor overhead (1.07%). In particular, we focus on the slow





**Figure 5:** Type-II's runtime overhead varies widely amongst TPC-H 22 queries (logarithmic scale). HEDB achieves  $2.49\times$  speedup on average.



**Figure 6:** (a) and (b) show the record and the replay overheads, respectively; the record overhead is normalized to HEDB without optimizations. (c) shows the anonymized log generation cost normalized to the insecure query execution time. Y-axes of (b) and (c) use a logarithmic scale.

secure queries ( $10\times$  slower than insecure baselines), whose average overhead is  $7.49\%$ .

**Storage overhead.** HEDB's logs introduce moderate storage overhead. The corresponding logs for TPC-H (scale factor = 1.0), which occupies 5,523 MB of encrypted data, results in log files of 20,004 MB ( $3.62\times$ ) in size. After compression with gzip, the total size is reduced to 1,853 MB ( $9.26\%$  fraction). The compression is very effective because many log entries appear multiple times. Should storage quota be a concern, logs can be periodically truncated.

### 6.3.4 Replay-based Maintenance Overhead

**Query re-execution overhead.** DBAs often need to re-execute user queries to understand their behavior and check if proposed fixes take effect. HEDB's logs allow for faster query debugging, as they preserve the input-output relationship, eliminating all de/encryptions in re-execution for configuration and functional bugs. Figure 6b demonstrates the TPC-H query replaying overhead, showing that HEDB's log-based replay is  $3.96\times$  faster than Ops-based replay (by honestly calling operators), saving the DBAs time and effort. Nevertheless, replay still incurs  $5.11\times$  slowdown compared with the insecure baselines. To debug performance bugs, DBAs can enable HEDB's delay simulation feature, which maintains the same query performance as the real queries.

**Anonymized log generation overhead.** We evaluate HEDB's log anonymization, which transforms an authenticated log into an anonymized log. We measure the anonymized log generation time and present the results in Figure 6c. HEDB's optimizations, such as warm start for KLEE and constraint cache for Z3 (see § 5.2), result in a significant speedup of  $12\times$  to  $216\times$  on an 8-core VM. Specifically, HEDB's techniques

| Type    | Operation     | Proportion | KLEE (w/o fork) | KLEE (w/ fork) | Z3   |
|---------|---------------|------------|-----------------|----------------|------|
| Integer | comparison    | 47%        | 0.71            | 0.06           | 0.12 |
|         | computation   | 40%        | 0.70            | 0.05           | 0.12 |
|         | aggregation   | 13%        | 2.81            | 2.15           | 0.13 |
| String  | comparison    | 70%        | 0.77            | 0.12           | 0.12 |
|         | substring     | 10%        | 0.71            | 0.06           | 0.12 |
|         | concatenation | 10%        | 0.72            | 0.07           | 0.12 |
|         | search (LIKE) | 10%        | 1.25            | 0.61           | 0.14 |
| Time    | comparison    | 87%        | 0.74            | 0.10           | 0.12 |
|         | extraction    | 12%        | 2.08            | 1.41           | 0.19 |

**Table 6:** Log anonymization cost (in seconds) using KLEE and Z3.

improve KLEE constraint collection efficiency from 5 days to 2.7 hours, and reduced Z3 log generation time from 2 hours to 25 seconds. It is worth noting that Q18 is not supported because it processes floating-point numbers only, which HEDB currently does not support.

To assess the efficiency of our used tools (KLEE and Z3), we estimate the time required by each operator and report the worst-case time in Table 6. For KLEE-based concolic execution, aggregation operators like MIN take longer as these operators batch many items, but cost only  $\approx 0.03s$  per item when amortized. String operator "LIKE" (using a regular expression library) and timestamp operator "EXTRACT" (using big integer division) were also time-consuming. We reimplemented the division in EXTRACT to reduce it from 3.17s to 2.08s. Z3's constraint solving time depends on the number of constraints and symbolic variables. As a result, we found that only a few constraints exist in HEDB's operators.

## 7 Discussion

This section discusses several issues that HEDB currently does not address but are worth exploring as future work.



**Enforcing deterministic replay.** HEDB relies on record-and-replay (R&R) of operator invocations to reproduce EDB issues. However, due to the non-deterministic nature of concurrency, HEDB does not support debugging queries with non-determinism, e.g., concurrent writes. While providing deterministic R&R frameworks would be essential for bug reproduction, it is orthogonal to our work. Alternatively, DBaaS providers may also consider deterministic databases [46].

**Fully supporting query rewriting.** DBAs need to help rewrite user queries for tuning (see Table 5). However, HEDB does not support all query rewriting because allowing unseen invocations could raise security issues. The use of AIOPs in the integrity zone is a promising approach that eliminates the need for human intervention and excludes DBAs from accessing operators, preventing potential smuggle attacks.

**More flexible operator troubleshooting.** If user-defined masking rules are too restrictive, HEDB’s anonymized replay may hinder the reproduction of bugs triggered by certain values, such as division by zero. We aim to develop more flexible masking rules that can disclose more operator bugs.

**Examining metadata log privacy.** Database logs are heavily used for DBAs to diagnose DBMS issues [38, 52]. EDBs are no exception. In HEDB, the record-and-replay logs are either encrypted or anonymized. However, various metadata logs exist in the integrity zone and might leak privacy. According to our investigation, a DBaaS provider typically collects the following logs (and potentially more):

- *Syslogs*: errors and exceptions of the DB processes.
- *Operation logs*: operations from all SQL clients/DBAs.
- *Trace logs*: internal exception logs for DBMS engines.
- *WAL logs*: transactions that make changes to the DB.
- *Performance logs*: environmental resource status, including CPU, disk, and network I/O statistics.

In the future, we plan to examine their leakage profiles.

**Porting to other architectures.** The current prototype uses ARMv8.4 S-EL2 for fast mode switch. We plan to port HEDB to other VM-based confidential computing platforms such as AMD SEV [13], Intel TDX [9] and ARMv9 CCA [36], and explore optimization techniques for these architectures.

## 8 Related Work

**Encrypted databases (EDBs).** There is an increasing interest in EDBs from academia [15, 17, 26, 42–44, 48] and industry [14, 30, 50]. Type-I EDBs [17, 44, 45] lack DBA maintenance. Crypto-based Type-II EDBs [42, 43] lack full SQL support. TEE-based Type-II EDBs [15, 48] suffer from smuggle attacks. Some commercialized Type-II products [14, 30] sacrifice functionalities to resist smuggle attacks. Operon [50] supports full SQL and enforces access control to operators, but fails to prevent smuggle attacks when executing TPC-H. In contrast, HEDB achieves full SQL, DBA maintenance and

interface security by introducing a dual-mode EDB design.

**EDB attacks.** A long line of studies has discussed the leakage attacks of EDB systems, including ordering, distribution, volume, access patterns, and frequency analysis [27–29, 32, 39]. These types of leakage can be vulnerable to passive attackers who attempt to recover the original data with sophisticated background knowledge [29, 32, 39]. On the other hand, active attacks that breach ordering without user authorization are further discussed in [27]. We devise a new active attack—smuggle attacks—which requires zero background knowledge and is challenging to detect.

**Analytical privacy processing.** Monomi [47] splits client-server query execution to support TPC-H over encrypted data. Monomi requires a client-side computational platform, while HEDB executes the full query on an untrusted cloud.

**Record-and-replay (R&R) for databases.** R&R is a well-studied technique in database systems. FoundationDB [54] uses R&R for deterministic distributed transactions. Zhang et al. [53] adopts an R&R framework for ACID testing. HEDB confines the misbehaviors of distrustful DBAs with R&R.

**Privacy-preserving debugging systems.** Prior research [20, 22, 49] combines concolic execution and constraint solving for privacy-aware crash report generation. Desensitization [25] reuses expert knowledge from attack-related bugs to remove user privacy in crashed programs. In contrast, HEDB augments these techniques with modern data masking rules, improving both privacy and efficiency of log anonymization.

## 9 Conclusion

Encrypted databases (EDB) are the holy grail of database security. HEDB is a novel EDB design that achieves interface security yet preserves database maintainability. Execution Mode prevents illegal invocations to operators while Maintenance Mode allows untrusted DBAs maintenance. HEDB introduces several key techniques such as authenticated replay and anonymized replay. The source code of HEDB is publicly available at <https://github.com/SJTU-IPADS/HEDB>.

## Acknowledgments

We sincerely thank our shepherd, Manuel Costa, and the anonymous reviewers of OSDI 2023 for their constructive comments. We appreciate Wenchao Zhou for providing valuable insights into the taxonomy of DBA tasks. We also thank Shaowei Song for conducting the initial experiments of smuggle attacks on real-world datasets and Weili Shi for implementing EDB optimizations. This work was supported by Alibaba Group through Alibaba Innovative Research Program, and in part by National Key Research and Development Program of China (No. 2020AAA0108500), National Natural Science Foundation of China (No. 62132014, 61925206, U19A2060), STCSM (No. 21511101502). Yubin Xia ([xiayubin@sjtu.edu.cn](mailto:xiayubin@sjtu.edu.cn)) is the corresponding author.

## References

- [1] Amazon Aurora. <https://aws.amazon.com/rds/aurora/>.
- [2] Azure SQL Database - Dynamic data masking. <https://learn.microsoft.com/en-us/azure/azure-sql/database/dynamic-data-masking-overview>.
- [3] BUG #15727: PANIC: cannot abort transaction 295144144, it was already committed. <https://www.postgresql.org/message-id/15727-0be246e7d852d229%40postgresql.org>.
- [4] Cost of a data breach 2022. <https://www.ibm.com/reports/data-breach>.
- [5] Data masking using AWS DMS. <https://aws.amazon.com/cn/blogs/database/data-masking-using-aws-dms/>.
- [6] The digitization of the world from edge to core. <https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf>.
- [7] Google Cloud SQL. <https://cloud.google.com/sql>.
- [8] Hospital Inpatient Discharges (SPARCS De-Identified): 2012. <https://health.data.ny.gov/Hospital-Inpatient-Discharges-SPARCS-De-Identified/u4ud-w55t>.
- [9] Intel Trust Domain Extensions. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>.
- [10] intersect seem not working correctly. <https://github.com/dimitri/prefix/issues/13>.
- [11] Supporting intel sgx on multi-socket platforms. <https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html>.
- [12] What the historic leak of swiss banking records reveal. <https://mg.co.za/business/2022-02-22-what-the-historic-leak-of-swiss-banking-records-reveal/>.
- [13] AMD. AMD Secure Encrypted Virtualization (SEV). <https://developer.amd.com/sev/>.
- [14] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure SQL database always encrypted. In *Proceedings of the ACM SIGMOD Conference*, 2020.
- [15] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2013.
- [16] Arvind Arasu, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. Integrity-based attacks for encrypted databases and implications. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.
- [17] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the ACM SIGMOD Conference*, 2011.
- [18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [19] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [20] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
- [21] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [22] James A. Clause and Alessandro Orso. Camouflage: automated anonymization of field data. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.
- [23] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer, 2008.
- [24] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [25] Ren Ding, Hong Hu, Wen Xu, and Taesoo Kim. DESENSITIZATION: privacy-aware and attack-preserving crash report. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.
- [26] Benny Fuhry, Jayanth Jain H. A, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. 2021.
- [27] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *Proceedings of the IEEE Symposium on Security and*

- Privacy (S&P)*, 2017.
- [28] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.
- [29] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.
- [30] Liang Guo, Jinwei Zhu, Jiayang Liu, and Kun Cheng. Full encryption: An end to end encryption mechanism in gaussdb. 2021.
- [31] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. IEEE Computer Society, 2002.
- [32] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O’neill. Generic attacks on secure outsourced databases. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
- [33] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can’t live with ’em, can’t live without ’em. In *International Conferences on Information Science and System*, 2008.
- [34] Michael A. Kozuch, Michael Kaminsky, and Michael P. Ryan. Migration without virtualization. In Armando Fox, editor, *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.
- [35] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twinvisor: Hardware-isolated confidential virtual machines for ARM. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
- [36] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.
- [37] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. Floating-point symbolic execution: a case study in n-version programming. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.
- [38] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. Diagnosing root causes of intermittent slow queries in large-scale cloud databases. 2020.
- [39] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [40] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Riccardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and validating database system administration. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.
- [41] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [42] Rishabh Poddar, Tobias Boelter, and Raluca A. Popa. Arx: An encrypted database using semantically secure encryption. 2019.
- [43] Raluca A. Popa, Catherine M. S. Redfield, Nikolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
- [44] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [45] Pedro S. Ribeiro, Nuno Santos, and Nuno O. Duarte. Dbstore: A trustzone-backed database management system for mobile applications. In *International Conference on E-Business and Telecommunication Networks*, 2018.
- [46] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD Conference*, 2012.
- [47] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nikolai Zeldovich. Processing analytical queries over encrypted data. 2013.
- [48] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *Proceedings of the Privacy Enhancing Technologies Symposium (PETs)*, 2019.
- [49] Rui Wang, XiaoFeng Wang, and Zhuowei Li. Panalyst: Privacy-aware remote error analysis on commodity software. In *Proceedings of the USENIX Security Symposium*, 2008.
- [50] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. Operon: An encrypted database for ownership-preserving data management. 2022.
- [51] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, 1982.
- [52] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transac-

tional databases. In *Proceedings of the ACM SIGMOD Conference*, 2016.

- [53] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [54] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the ACM SIGMOD Conference*, 2021.

## A Appendix

### A.1 Attacking Encrypted Data Types

1. *Integer*: The DBA leverages arithmetic operators (+, −, ×, ÷) and comparison operators (>, =) to construct an encrypted arithmetic progression which assists in recovering the original integers, as described in § 2.5.
2. *Decimal*: Like Integer, a DBA can construct ciphertexts equal to 1.0 using arithmetic operators. With +, 10.0 can be derived and further help construct 0.1 by dividing 1.0 by 10.0. Similarly, 0.01 and 0.001 can also be recovered. Using these pivot values, 32-bit Real and 64-bit Double can be recovered in terms of integral and fractional parts, respectively.
3. *Text*: Text does not support arithmetic operators. Still, a DBA can invoke the operator `substring(string, from, to)` which splits each character, by manipulating the encrypted integer arguments `from` and `to`. As the character has a finite domain (256 as defined in ASCII), once 256 different values are fulfilled, a DBA can infer the actual character and the original text.
4. *Time*: Because DBMSes support arithmetic operators such as +, −, < on 32-bit Date and 64-bit Time, these operators can be exploited to conduct full recovery.

### A.2 HEDB Query Execution Optimization

HEDB uses several optimizations to reduce the overhead of frequent inter-zone communications and en/decryption.

**Parallel Decryption (O1).** Data decryptions in an expression can be done in parallel as they do not depend on one another. HEDB uses a thread pool: when a new invocation arrives, the dispatcher seeks an idle thread and assigns a decryption task.

**Order-revealing Encryption (O2).** We observe from realistic workloads that encrypted texts have many comparisons, but only a few unique values. Also, ordering is revealed during query execution. We thus insert the integer order into each encrypted text’s header. HEDB utilizes the embedded order to

compare two encrypted text values, avoiding decryption.

**Expression Evaluation (O3).** User queries might contain complex expressions. For instance, TPC-H Q1 contains `SUM(l_extendedprice * (1 - l_discount))`. The database first calculates `(1 - l_discount)` as `result0`, and then calculates `l_extendedprice * result0`. In total, 3 decryptions and 2 encryptions are performed. To reduce the redundant en/decryptions, HEDB parses the whole expression, leading to 2 decryptions and 1 encryption. Aggregations (e.g., SUM, AVG) are also optimized using expressions.

## B Artifact Appendix

### B.1 Abstract

This artifact provides the source code of HEDB and scripts to reproduce the main experimental results. To reproduce the results in § 6, we provide instructions to build binaries and run experiments. The source code of HEDB can be retrieved from a public open-source repository under the Mulan Permissive Software License v2. Although the scripts target our testbed, readers can port them to other platforms. For those interested in using HEDB in their own research, we recommend using the `main` branch of our repository, which would be maintained by members of the Institute of Parallel and Distributed Systems.

### B.2 Scope

The artifact contains instructions and scripts for reproducing Figure 5 and Figure 6 that support the following four claims:

- **Claim-1:** HEDB’s optimizations speed up Type-II EDB.
- **Claim-2:** HEDB’s record overhead is low and acceptable.
- **Claim-3:** HEDB’s replay overhead is much faster than operator-based replay.
- **Claim-4:** HEDB’s optimizations boost the anonymized log generation speed.

### B.3 Hosting

The artifact is publicly available at our GitHub repository:

```
git clone https://github.com/SJTU-IPADS/HEDB
git checkout main
```

### B.4 Contents

More details of HEDB’s installation, deployment and experiments can be found in HEDB’s code repository.







# LVMT: An Efficient Authenticated Storage for Blockchain

Chenxing Li    Sidi Mohamed Beillahi<sup>†</sup>    Guang Yang    Ming Wu    Wei Xu<sup>‡</sup>    Fan Long<sup>†</sup>

*Shanghai Tree-Graph Blockchain Research Institute*  
*University of Toronto<sup>†</sup>    Tsinghua University<sup>‡</sup>*

## Abstract

Authenticated storage access is the performance bottleneck of a blockchain, because each access can be amplified to potentially  $O(\log n)$  disk I/O operations in the standard Merkle Patricia Trie (MPT) storage structure. In this paper, we propose a multi-Layer Versioned Multipoint Trie (LVMT), a novel high-performance blockchain storage with significantly reduced I/O amplifications. LVMT uses the authenticated multipoint evaluation tree (AMT) vector commitment protocol to update commitment proofs in constant time. LVMT adopts a multi-layer design to support unlimited key-value pairs and stores version numbers instead of value hashes to avoid costly elliptic curve multiplication operations. In our experiment, LVMT outperforms the MPT in real Ethereum traces, delivering read and write operations six times faster. It also boosts blockchain system execution throughput by up to 2.7 times.

## 1 Introduction

Blockchains that provide decentralized, robust, and programmable ledgers at an internet scale have recently gained increasing popularity across various domains, including financial services, supply chain, and entertainment. For example, smart contracts built on blockchain systems now manage digital assets worth tens of billions of dollars [3].

Early classical blockchain systems like Bitcoin [36] and Ethereum [17] have serious performance bottlenecks in their consensus protocols, which limit the system throughput at under 30 transactions per second. Nevertheless, recent technique evolutions on consensus and peer-to-peer network protocols [8, 22, 23, 26, 29, 31, 33, 35, 37, 44, 45, 51, 52] have driven the achievable blockchain throughput to more than thousands of transactions per second. Consequently, transaction execution, which is dominated the storage access, has emerged as the new system bottleneck. Our investigation (see Sec. 6)

shows that 81% of transaction execution time is consumed at the storage layer.

This inefficiency in the blockchain storage layer originates from the requirement for *authentication*. A standard permission-less blockchain system has two types of blockchain nodes: the full nodes and the light nodes. A full node synchronizes and executes all transactions, maintaining the blockchain ledger state. A light node (client) only synchronizes the block headers, excluding transactions and the blockchain ledger state. Blockchain ledger states take the form of key-value pairs. When a light node needs to ascertain the value of a given key, it queries a full node. However, since blockchain nodes are permissionless, light nodes should not trust the responses from full nodes. Therefore, the blockchain protocol requires the block proposer to compute a commitment (termed the *state root*) for the latest ledger state and insert it into the proposed block header. A block header with an incorrect commitment is deemed invalid. When responding to the queries from light nodes, a full node can generate proofs corresponding to the commitments to convince the queriers. This leads to the naming of the ledger state as *authenticated*.

Typically, authenticated storage employs the Merkle Patricia Trie (MPT) [5] structure, a specific variant of the Merkle tree. Each leaf node in an MPT stores a value, and the path from the root to the leaf node corresponds to the key of the stored value. Each inner node in the MPT stores the crypto hash of the concatenated contents of all its children. The MPT's root hash serves as the commitment of the blockchain state for authentication.

Unfortunately, this authentication comes with a heavy performance price. Modifying a key-value pair in the state requires an MPT to update hashes of all nodes along the path from the corresponding leaf node to the root. If not cached, each state update operation could be amplified to  $O(\log n)$  disk I/O operations, where  $n$  represents the storage size. Note that even a basic payment transaction involves at least two ledger

state updates, – decreasing the sender’s balance and increasing the receiver’s. As the throughput of recent blockchains approaches thousands of transactions per second, it is not surprise that storage becomes the new bottleneck.

This paper presents LVMT, a novel high-performance authenticated storage framework with significantly reduced disk I/O amplifications. LVMT achieves high efficiency by integrating a multi-level Authenticated Multipoint evaluation Tree (AMT) and a series of append-only Merkle trees. AMT is a cryptographic vector commitment scheme that can update commitment (i.e., the hash root) in constant time [50]. Despite its constant commitment update time, there are several key challenges to address when incorporating AMT into the LVMT design.

The first challenge arises from the expensive elliptic curve multiplication operations employed by the AMT commitment update algorithm. A naive approach would paradoxically result in a slower state update operation on the AMT than the MPT, despite the theoretically reduced amplification. LVMT addresses this challenge with its novel *key-versioned-value* design. It assigns each key a version, incrementing as the value evolves. Rather than storing key-value pairs in the AMT, LVMT employs AMT to keep key-version pairs and uses Merkle Trees to maintain an append-only authenticated list of *key-version-value triples*. Thus, every update in LVMT results in an increment of the stored version within the AMT. Since the AMT algorithm multiplies a precomputed elliptic curve point with the difference between the old value and the new value (i.e., one for a version increment) during a commitment update, LVMT effectively eliminates the expensive multiplication. Also, because the key-version-value triple list is append-only, LVMT only needs to construct these Merkle Trees once during the block commit time, and therefore the process is very efficient.

The second challenge emerges from AMT’s limitation in supporting the necessary bit-depth for blockchain state keys. An AMT with  $k$ -bit key-space requires public parameters with  $2^k$  elliptic curve points. To enable efficient update, the AMT also requires pre-computation and caching of elliptic curve points proportional to the public parameters’ size. Even for a modest 32-bit key-space, the precomputed metadata size would exceed 256 GB, which is untenable, given that blockchain ledger keys typically comprise 256 bits. To address this challenge, LVMT operates with a novel *multi-level multi-slot structure*, integrating multiple AMTs. Each AMT in this structure has a 16-bit key-space, and the structure can automatically generate a sub-AMT on the next level to accommodate keys-version pair with collided prefix. Since collisions are rare after the first level and creating sub-AMT will make subsequent access more expensive, LVMT also makes each

entry in AMTs contain five slots. Therefore expansion to the next level only occur when more than five collisions arise.

The third challenge lies in the costly maintenance of proof generation metadata. While updating the root hash for AMT incurs constant time, maintaining the proof generation metadata still requires  $O(\log n)$  time and triggers the same degree of I/O amplifications as MPT. LVMT confronts this issue with a *proof sharding technique*, which distributes the proof generation metadata to multiple nodes. In LVMT, each full node only maintains the proof generation metadata for a shard of the blockchain state (e.g., keys sharing the same 4-bit prefix). Our observation reveals that there are typically thousands of full nodes in a production blockchain, and it’s unnecessary for all nodes to maintain proof generation capabilities for all key-value pairs in the total state. Even sharded, for any part of the state, there will still be enough nodes serving proof generation requests from light clients. Within the current Ethereum ecosystem, most light nodes access full nodes from specialized providers, such as Infura, who operate several full nodes to balance query workload. By maintaining proof shards across their nodes, these providers can efficiently generate proof for any key. Note that unlike other sharding designs [18, 29, 34, 51, 53], our proof sharding does not alter the essential obligation of each full node to synchronize and validate blocks, process all transactions, and accurately maintain the state root, thereby preserving security.

We have implemented LVMT [1] and integrated it into Conflux [2, 33], an open-sourced high-performance blockchain production with smart contract support. We evaluated LVMT against OpenEthereum’s MPT implementation, RainBlock’s MPT structure [40], and LMPTs [20], considering both stand-alone read/write workload and end-to-end blockchain processing tasks. Our results show that LVMT achieves up to 10x higher throughput on random state read/write operations. When integrated end-to-end with a high-performance blockchain, LVMT achieves up to 2.7x higher throughput for simple payment transactions and up to 2.1x higher throughput for ERC20 [41] token transfer transactions. This boost in performance stems from the considerable reduction in disk I/O amplifications. In terms of amplification, LVMT performs up to 4.1x better than MPT on read operations and up to 8.2x better on write operations.

## 2 Background

In this section, we recall some background on cryptographic concepts that our system builds on. In particular, we introduce the cryptographic building blocks of the Authenticated Multipoint evaluation Tree (AMT) [50], an efficient vector commitment protocol.

**Notations:** We denote  $[n]$  as the integers in  $\{x \in \mathbb{Z}^+ | 1 \leq x \leq n\}$ .  $\mathbb{G}$  signifies an elliptic curve group and symbols in upper cases like  $G, P$  represent elements in the elliptic curve groups.  $\mathbb{Z}_p$  refers to an additive group with order  $p$ .

## 2.1 Authenticated Storage in Blockchain

In a standard permission-less blockchain system, blockchain nodes can be distinguished into two types: full nodes and light nodes. A full node synchronizes and executes all transactions, maintaining the blockchain ledger state accordingly. A light node (client) synchronizes only the block headers, excluding transactions and blockchain ledger state.

When a full node proposes a new block, it is required to execute transactions in that block and incorporate the commitment of the post-execution ledger state into the block header. The node keeps a write-back cache during transaction execution, committing all modifications to the storage after executing all transactions in a block. The authenticated storage needs to provide two interfaces to the execution engine:

- $\text{Get}(k) \rightarrow v$ : Retrieves the value  $v$  associated a given key  $k$ .
- $\text{Set}(\{(k, v)_i\}, e) \rightarrow \text{comm}$ : Flushes a series of key-value pairs  $(k, v)$  to the storage with block number  $e$ , obtaining the commitment  $\text{comm}$  of the ledger state after changes.

When a light node wants to know the value of a specific key, it queries a full node, expecting a response of the value along with proof with respect to the ledger commitment. The light client examines whether the commitment exists within the set of verified valid commitments, then checks the validity of the associated proof. So the authenticated storage must provide two algorithms for proof generation and verification:

- $\text{Respond}(k) \rightarrow (v, \pi, \text{comm})$ : Returns the value  $v$  of key  $k$  with proof  $\pi$  with respect to the most recent commitment  $\text{comm}$ .
- $\text{Verify}(k, v, \pi, \text{comm}) \rightarrow \text{true/false}$ : Validates the response from the full node.

## 2.2 Elliptic Curve Group

The *elliptic curve group* plays a fundamental role in various cryptographic protocols. This group conducts an additive operation over points on an elliptic curve, such as  $\{(x, y) \in \mathbb{Z}_q^2 | y^2 = x^3 + x + 7\}$ , where  $q$  is a large prime number. An infinite point is included as the identity element. The operation  $a \cdot P$  represents  $P$  added to itself  $a$  times within the group, where  $a$  is a positive integer, and  $P$  is a point on the curve. An elliptic curve group is characterized by a *starting point*  $G$ , from which a sequence of points  $G, 2 \cdot G, 3 \cdot G, \dots$  can be generated. If the elliptic curve group is cryptographically secure, this sequence exhibits the following properties:

1.  $n \cdot G$  is periodic in  $n$ , with the period being a large prime integer  $p$ , i.e.,  $n \cdot G = (n + p) \cdot G$ ;
2. For a randomly selected  $n$ , deriving  $n$  from  $n \cdot G$  is computationally unfeasible.

## 2.3 KZG Commitment

Kate et al. proposed KZG polynomial commitment protocol [28], enabling someone to commit a polynomial function  $f$  to a commitment, and prove the value  $f(x)$  of any given position  $x$  with respect to that commitment.

The KZG commitment protocol is built on a bilinear map. Consider  $G_1$  and  $G_2$  as the starting points of two elliptic curve groups  $\mathbb{G}_1, \mathbb{G}_2$  respectively, each with the same group order  $p$ . The bilinear map  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  is homomorphic such that the equation  $e(a \cdot G_1, b \cdot G_2) = ab \cdot e(G_1, G_2)$  holds for any  $a, b \in \mathbb{Z}_p$ . Here,  $\mathbb{G}_T$  denotes another group of the same order  $p$ . BLS12-381 [14] from BLS families [9] and BN254 [11] from BN families [10] are widely-used deployed systems implementing bilinear maps. The groups  $\mathbb{G}_1$  and  $\mathbb{G}_2$  are elliptic curve groups of order  $p$ , and  $G_1$  and  $G_2$  are their perspective starting points.

For a given polynomial function  $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$  of degree  $n$ , the KZG commitment assumes a series of public parameters  $\tau \cdot G_1, \tau^2 \cdot G_1, \tau^3 \cdot G_1, \dots, \tau^n \cdot G_1$  in a trusted setup and commits function  $f$  to  $C := f(\tau) \cdot G_1$ . The public parameters are generated by a trusted party using a random  $\tau$ , which is forgotten after generation. Secure multi-party computation protocols [15, 16, 25] enable multiple participants to collaboratively generate these public parameters, ensuring that no participant can ascertain the exact value of  $\tau$ .

For any index  $i \in \mathbb{Z}_p$ , the expression  $x - i$  should divide  $f(x) - f(i)$ . This suggests that  $h_i(x) := \frac{f(x) - f(i)}{x - i}$  is indeed a polynomial. Hence, the proof  $\pi$  of  $f(i)$  is defined as  $h_i(\tau) \cdot G_1$ . Given that  $h_i(x)$  is a polynomial, the prover can compute the coefficients of  $h_i(\tau)$ . Thus,  $h_i(\tau) \cdot G_1$  forms a linear combination of the public parameters with known coefficients. The prover can compute it in a short time. A verifier, querying  $i$  with answer  $y = f(i)$  and proof  $\pi := h_i(\tau) \cdot G_1$ , can verify the proof by checking if

$$e(\pi, (\tau - i) \cdot G_2) = e(C - y \cdot G_1, G_2).$$

If the proof  $\pi$  is correctly constructed, the check must pass because

$$\begin{aligned} e(\pi, (\tau - i) \cdot G_2) &= (h(\tau) \cdot (\tau - i)) \cdot e(G_1, G_2) \\ &= e((f(\tau) - f(i)) \cdot G_1, G_2) \\ &= e(C - y \cdot G_1, G_2). \end{aligned}$$



If  $f(i) \neq y$ ,  $h(x)$  becomes a fraction, making it difficult to find a proper proof without knowing  $\tau$ . Kate et al. proved the binding property of this protocol [28].

The KZG commitment also supports the proof of a batch of positions. To prove that  $f(x)$  equals to 0 at a set of positions  $S$ , the proof  $\pi$  is constructed by  $\frac{f(\tau)}{\prod_{i \in S} (\tau - i)} \cdot G_1$ .

A vector commitment scheme can be built with KZG commitment by converting a vector  $\vec{a}$  to a polynomial function  $f$  by Lagrange interpolation. Formally, for an input vector  $\vec{a}$  with  $n$  elements, the interpolated function  $f$  is defined by  $f(x) = \sum_{i=1}^n a_i \cdot I_{i,n}(x)$ , where  $a_i$  is the  $i$ -th element of  $\vec{a}$  and  $I_{i,n}(x)$  is a Lagrange function that satisfies  $I_{i,n}(i) = 1$  and  $I_{i,n}(x) = 0$  for  $x \neq i$  and  $1 \leq x \leq n$ .

When updating the value at position  $i$  from  $a_i$  to  $a'_i$ , the corresponding commitment  $C$  can be simply updated to

$$C' := C + (a'_i - a_i) \cdot I_{i,n}(\tau) \cdot G_1. \quad (1)$$

If the prover caches results  $I_{i,n}(\tau) \cdot G_1$  for all  $i$ , updating commitment requires only one multiplication and one addition on the elliptic curve  $\mathbb{G}_1$ , which takes  $O(1)$  time.

## 2.4 Authenticated Multipoint Evaluation Tree

Although the KZG commitment enables constant-time updates to the commitment  $C$ , it requires  $O(n)$  time to construct a proof for a given position or to maintain proofs for all positions. In a blockchain system, where the vector being committed to is frequently changing, the KZG commitment cannot generate proofs efficiently for queries with arbitrary indices  $i$ .

To address this issue, Alin et al. proposed the Authenticated Multipoint evaluation Trees (AMT) commitment protocol [50], which maintains auxiliary information of size  $O(n \log n)$  and can generate a proof in  $O(\log n)$  time.

Consider an example with  $n = 8 = 2^3$ . For an input vector  $\vec{a}$  with eight elements, AMT computes its Lagrange interpolation  $f(x)$  which satisfies  $f(i) = a_i$  for  $1 \leq i \leq 8$ . The function  $f(x)$  is then partitioned into two functions  $f_0(x)$  and  $f_1(x)$ . In the subset  $x \in [8]$ ,  $f_1(x)$  mirrors  $f(x)$  for even  $x$  and is zero otherwise, while  $f_2(x)$  mirrors  $f(x)$  for odd  $x$  and is zero otherwise. For values of  $x$  outside this subset,  $f_1(x)$  and  $f_2(x)$  are determined by Lagrange interpolation. Consequently,  $f(x)$  can be re-expressed as  $f(x) = f_0(x) + f_1(x)$ . AMT continues to subdivide  $f_0(x)$  recursively into two functions:  $f_{0,0}(x)$  and  $f_{0,1}(x)$ . Here,  $f_{0,0}$  mirrors  $f(x)$  for  $x \in \{4, 8\}$ , and  $f_{0,1}(x)$  mirrors  $f(x)$  for  $x \in \{2, 6\}$ . This recursive process of partitioning generates a full binary tree, where each node corresponds to a function. Each inner node's function is the sum of the function at its child nodes, and each leaf node is a multiplication of an identity Lagrange function because it mirrors  $f(x)$  at a single point  $x$ . For example,  $f_{0,0,1}(x) = a_4 \cdot I_{4,8}(x)$ .

Each inner node of the AMT is associated with two elements: 1) the KZG commitment of its corresponding function and 2) a batch proof for the indices at which the function is zero according to the partitioning process. Detailed definitions of these elements are provided in the appendix. When proving the value of a given entry, e.g.,  $a_4$ , the prover finds the path from the root to the corresponding leaf node:  $f(x) \rightarrow f_0(x) \rightarrow f_{0,0}(x) \rightarrow f_{0,0,1}(x)$ . It then iteratively decomposes functions along this path to express  $f(x)$  into as a sum of four components:  $f_1(x) + f_{0,1}(x) + f_{0,0,0}(x) + f_{0,0,1}(x)$ . The prover then outputs the associate commitments for  $f_1(x)$ ,  $f_{0,1}(x)$ , and  $f_{0,0,0}(x)$ , alongside their batch proofs demonstrating these functions equal to zero at  $x = 4$ . The verifier checks the correctness of these batch proofs and the consistency among commitments: whether the sum of commitments for  $f_1(x)$ ,  $f_{0,1}(x)$ ,  $f_{0,0,0}(x)$ , and  $f_{0,0,1}(x) = a_4 \cdot I_{4,8}(x)$  equals to the commitment for  $f(x)$ .

Updating an entry in the AMT involves traversing from the root to the leaf corresponding and updating the associate elements along this path. The remaining are not affected, enabling AMT to maintain the proofs in  $O(\log n)$  time.

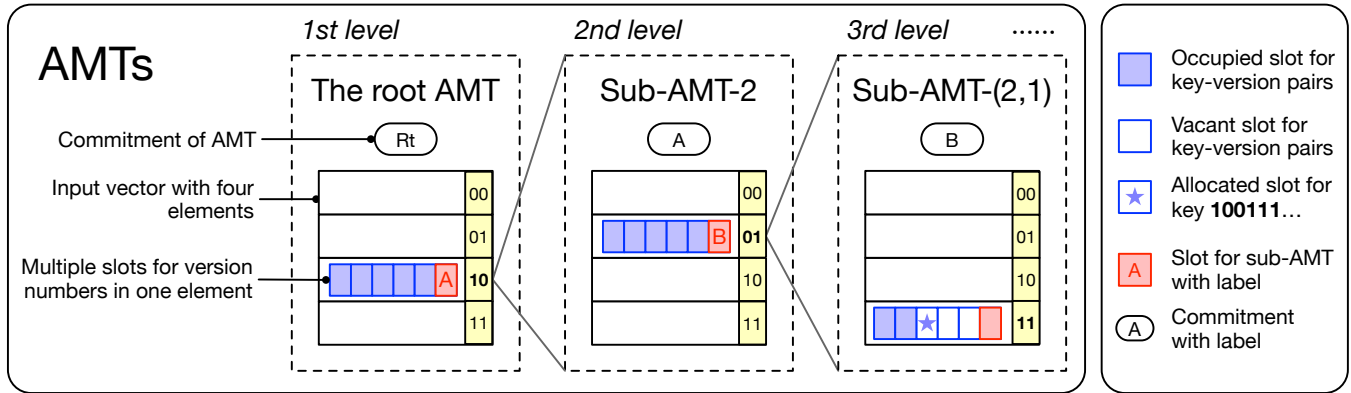
The nodes of the AMT serve as *auxiliary information* for generating proofs only. In a blockchain system, a miner without serving client queries may discard this auxiliary information and only maintain the commitment, which can be updated in constant time.

## 3 Overview

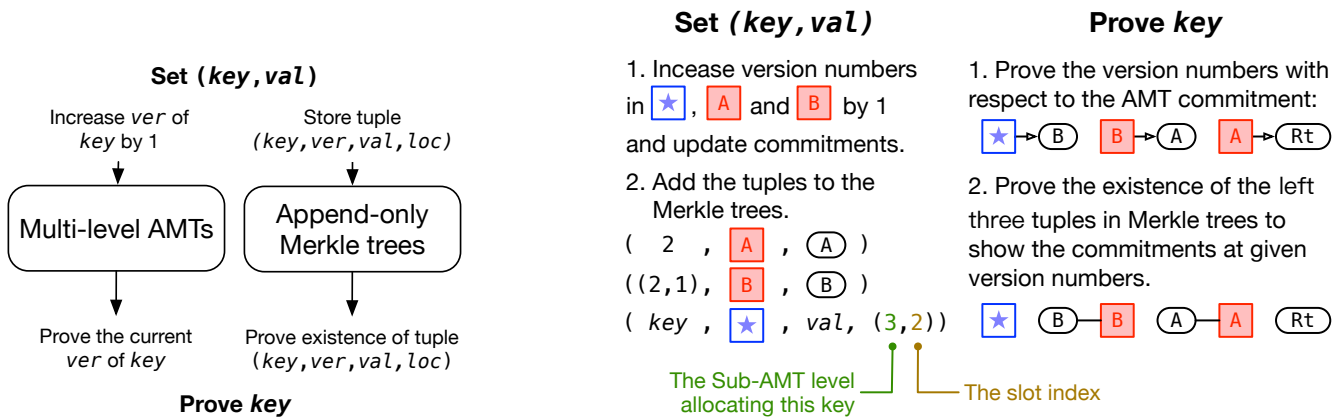
Recent works [32, 42] have shown that the majority of transactions execution time is spent on operations that access the blockchain state. For instance, a profiling experiment [32] shows that read and write operations to the blockchain state account for more than 67% of the execution time for the transaction executing the transfer function of ERC-20 smart contract [4, 41]. In this section, we present an overview of how LVMT tackles this problem. In particular, we propose a new authenticated storage system to reduce the amplification of read and write operations that access the blockchain state.

Our proposed system is based on AMT since it has an ideal time complexity, i.e., constant cost in updating the commitment. In particular, our proposed system solves several challenges to implement an efficient blockchain storage system using AMT:

First, although AMT costs constant time in updating the commitment, the constant ratio is large for a blockchain system. Table 1 shows the result of a micro-benchmark carried on an Intel i9-10900K CPU machine. It shows the time cost for basic cryptographic operations. Note that an elliptic curve multiplication takes about 0.1 ms, which is even much slower



(a) Multi-level AMTs



(b) Versioned key-value database

(c) Maintenance and proving on Multi-level AMTs

Figure 1: LVMT architecture.

| Pairing engines | BLS12-381 | BN254 |
|-----------------|-----------|-------|
| Addition        | 0.68      | 0.34  |
| Multiplication  | 169       | 92    |

Table 1: Time cost of operations over the primary curve  $\mathbb{G}_1$  of pairing functions ( $\mu s$ ).

than an updating operation in MPT.

Second, to support data with  $n$  maximum entries, AMT requires precomputed parameters in size of  $O(n \log n)$  and maintains auxiliary information in size of  $O(n \log^2 n)$ . Thus, AMT cannot support key-value pairs for an arbitrary-length bit string. As the size of the blockchain ledger state continues to grow, AMT is not a scalable solution.

Last, a blockchain system must consider the slowest node. Even if most miners do not need to maintain the auxiliary information for proof, the authenticated storage must guarantee the nodes for responding queries can keep up.

We propose the following techniques to resolve the challenges above. First, we design a versioned database that only stores the version number of keys in AMT, thereby avoiding

the elliptic curve multiplications. This design also supports arbitrary lengths of values, as they are not stored in AMT. Second, we extend AMT to multiple levels to accommodate version numbers for unlimited keys, making the AMT size relatively small to optimize cache for parameters. To support arbitrary key lengths and minimize deep updates in the multi-level hierarchy, we utilize key hashes to allocate slots for version numbers. Last, we introduce proof sharding to reduce the single node's cost in maintaining auxiliary information for proofs.

### 3.1 Versioned Key-value Database

We designed a versioned authenticated storage to avoid multiplication on the elliptic curve during commitment updates. As shown in Figure 1b, the multi-level AMTs store key-version pairs, which only serve to identify the recent version number of a key. LVMT accumulates the key-version-value tuples in an append-only authenticated data structure consisting of a series of Merkle trees, with each block constructing one Merkle tree from the key-version-value tuples for value changes in

that block.

Imagine a scenario where the blockchain processes a block, setting a key-value pair  $(key, val)$ . LVMT first locates the corresponding entry of  $key$  in the multi-level AMTs to increment the stored version number by one. Assume the new version number for  $key$  is  $ver$ . LVMT then appends a new tuple  $(key, ver, val, loc)$  to the Merkle tree being constructed for the block. Here,  $loc$  is a tuple  $(level, slot)$  that records the level and slot in the multi-level AMTs where the key's version is located. The construction cost of a Merkle tree is linear with the number of version tuples. Once constructed, the Merkle tree for a block remains immutable, except for garbage collection of obsolete nodes. As the blockchain is append-only, the list of these Merkle trees is also append-only.

When generating a proof for a key-value pair  $(key, val)$ , LVMT first use the multi-level AMTs to prove the most recent version  $ver$  of the key  $key$ . It then uses Merkle trees to prove the existence of a tuple  $(key, ver, val, loc)$ . Since the roots of the Merkle trees are endorsed by the blockchain consensus protocol, light clients can trust that the Merkle trees are generated correctly without duplicate tuples having the same  $key$  and  $ver$ . As the location of the version slot is included in the version tuple of the key, the prover can not cheat by providing a version number proof of another slot.

Note that updating one element  $a_i$  to  $a'_i$  in an AMT requires computing  $(a'_i - a_i) \cdot I_{i,n}(\tau) \cdot G_1$  (equation 1), multiplying  $a'_i - a_i$  to the elliptic curve point  $I_{i,n}(\tau) \cdot G_1$ . In the versioned key-value database,  $a_i$  is essentially a version number and  $a'_i - a_i$  always equal 1. Thus, we eliminate an elliptic curve multiplication in each storage write, saving approximately 100  $\mu s$ .

Since the frequency of bumping version number is limited by the block generation rate, we can conserve the bits used for storing version number and store multiple version numbers in a one vector entry. For example, when employing BN254 as the underlying bilinear mapping parameter, each entry is an element in  $\mathbb{Z}_p$ , where  $p$  is a prime integer in  $(2^{254}, 2^{255})$ . This suggests that implies each entry can store at most 254 bits. In a blockchain system generating 10 blocks per second, the version number will not exceed  $2^{40}$  in 3000 years. So each entry can be divided into six slots with 40 bits as shown in Figure 1a.

## 3.2 Multi-level AMT

To make AMT scalable and allow it to store the version number for an unlimited number of keys, we introduce multi-level AMTs as shown in Figure 1a. The authenticated storage is initiated by one AMT as *the root AMT*. Each entry in the AMT contains several slots for storing version numbers. One slot in each entry is reserved for storing the version number

of the commitment hash of the sub-AMT, with the remaining slots utilized for key-value pairs.

Let  $k$  denote the height of the AMT. When allocating a slot for a new key, LVMT accesses the entry in the root AMT whose index aligns with the first  $k$  bits of the key hash. If this entry lacks a vacant slot, LVMT accesses the corresponding sub-AMT and locates the entry in the sub-AMT whose index matches the next  $k$  bits of the key hash. LVMT recursively visits the sub-AMTs to find a vacant slot for the new key. Figure 1a presents an example with  $k = 2$  for allocating a version slot for a key with hash  $100111\dots$ . As the first two bits of key hash are 10, LVMT accesses the entry with index 2 and attempts to find a vacant slot. Since all slots in the entry are occupied, LVMT proceeds to the corresponding sub-AMT-2. Picking the next two bits 01, it accesses the entry with index 1, and recursively visits the sub-AMT-(2,1) because there is no vacant slot again. Finally, LVMT finds the third slot at the third level being vacant and allocates this slot.

The commitment of a sub-AMT is treated similarly to a key-value pair, where the key represents the index of the sub-AMT and the value is the commitment. The Merkle trees not only store key-version-value tuples for standard key-value pairs, but they also store the tuples of the sub-AMT index, the version of the sub-AMT commitment, and the commitment hash.

Figure 1c illustrates how LVMT maintains the AMTs and Merkle trees when a block changes the key with hash  $100111\dots$ . LVMT first increments the version number for this key by one. This in turn alters the commitment of sub-AMT-(2,1), prompting LVMT to also increase the version number for the commitment (the slot labeled "B") by one. Recursively, the commitment of sub-AMT-2 is changed and the version number labeled "A" is updated. Finally, LVMT gets the updated commitment of the root AMT. LVMT appends the tuples of changed keys and commitments into the Merkle trees along with the normal tuple of the key-value pair.

When generating a proof for this key, LVMT finds the most recent version of tuples for sub-AMT-2, sub-AMT-(2,1) and this key. LVMT proves the existence of these tuples in Merkle trees and confirms the correctness of appeared version numbers with respect to their AMT commitments. When proving the non-existence of a key, LVMT affirms that all the possible slots for this key are vacant or have been allocated to other keys.

## 3.3 Proof Sharding

We recall that the AMT maintains a binary tree, where each node holds a commitment and a batch proof. Each input entry corresponds to a leaf in this tree. When generating a proof, AMT picks commitments and batch proofs from the siblings

of nodes along the path from this leaf to the root. Each node can be updated independently of the other nodes, facilitating the parallelization of tree maintenance. Each blockchain node can maintain a shard of the proof. It picks a subtree of the root AMT and takes responsibility for generating proofs for the leaves in this subtree, and the sub-AMTs extended from these leaves. Multiple blockchain nodes can collaboratively generate proof for any key. Similarly, the storage for the Merkle tree can be distributed to multiple nodes by the block number.

## 4 LVMT Design

Now we formally define LVMT, which utilizes a key-value database as a backend and maintains a tuple of key-value maps (KM, AM, MM, VM, LM) where KM stores the key-value pairs, AM stores the AMTs data structures, MM stores the Merkle trees, VM stores the version slots metadata for keys, and LM records the position of the most recent tuple for a key or a sub-AMT in the Merkle trees. LVMT decouples the data storage and data authentication: KM stores unauthentication data; AM and MM store the authenticated information; VM and LM store the metadata and indices for authenticated information. Each AMT in LVMT encompasses the following components:

- comm: the commitment of AMT;
- proof\_tree: the proof tree of AMT;
- leaves: a list of leaves; leaves[ $i$ ] denotes the leaf corresponding to the  $i$ -th element of the input vector. Each leaf comprises the two lists vers and keys. vers[0] stores the version number for the sub-AMT. vers[1] to vers[5] store the version numbers for the keys keys[1] to keys[5], respectively. Note that only vers contribute to the AMT commitment.

### 4.1 Interfaces to the Transaction Execution

LVMT provides the following two interfaces (instructions) for the blockchain execution layer:

- Get( $k$ )  $\rightarrow$  val: Reads the value val stored in  $k$ ;
- Commit( $\mathbf{W}, e$ )  $\rightarrow$  (aroot, hroot): Flushes the changed key-value pairs in  $\mathbf{W}$  with block number  $e$  and produces the commitment of LVMT.

These interfaces match the requirements from the blockchain execution engine introduced in Section 2.1. The execution engine uses Get to fetch data from the storage and LVMT simply loads the value correspondingly from KM.

The instruction Commit is invoked after the execution of a block. LVMT commits the key-value pairs  $\mathbf{W}$  using the procedure COM defined in Algorithm 1. The returned commitments will be filled in the block header. The commit returned values

---

**Algorithm 1** A procedure to compute a commitment. It takes a list of key-value pairs  $\mathbf{W}$  and a block number  $e$ , and returns the commitments aroot and hroot.

---

```

1: procedure COM( $\mathbf{W}, e$ )
2:    $\mathbf{M} \leftarrow []$ ;  $\mathbf{T} \leftarrow \{\}$ ;
3:   foreach ( $k, val$ ) in  $\mathbf{W}$ 
4:     ( $lv, tid_x, sid_x, ver$ )  $\leftarrow$  ComKV( $k, val$ );
5:      $\mathbf{M} \leftarrow (k, ver, val, lv, sid_x) :: \mathbf{M}$ ;
6:      $\mathbf{T} \leftarrow \{(lv, tid_x)\} \cup \mathbf{T}$ ;
7:    $i \leftarrow$  maximum  $lv$  in  $\mathbf{T}$ ;
8:   while  $i \geq 0$ 
9:     foreach ( $lv, tid_x$ ) in  $\mathbf{T}$  with  $lv = i$ 
10:      ( $C, ver$ )  $\leftarrow$  UpdComVer( $lv, tid_x$ );
11:       $\mathbf{M} \leftarrow (lv, tid_x, ver, comm) :: \mathbf{M}$ ;
12:      if  $lv > 0$ 
13:         $\mathbf{T} \leftarrow \{(lv - 1, \lfloor tid_x/n \rfloor)\} \cup \mathbf{T}$ ;
14:   foreach ( $k, ver, val, lv, sid_x$ ) in  $\mathbf{M}$  with index  $i$ 
15:     LM[ $k$ ]  $\leftarrow (e, i)$ ;
16:   foreach ( $lv, tid_x, ver, C$ ) in  $\mathbf{M}$  with index  $i$ 
17:     LM[ $(lv, tid_x)$ ]  $\leftarrow (e, i)$ ;
18:   Build merkle tree of  $\mathbf{M}$  and store inner nodes in MM;
19:   mroot  $\leftarrow$  Merkle root of  $\mathbf{M}$ ;
20:   hroot  $\leftarrow$  Merkle root of the mroot of all the commits;
21:   aroot  $\leftarrow$  AM[ $(0, 0)$ ].comm;
22:   return (aroot, hroot);
```

---



---

**Algorithm 2** A procedure to compute the commit of a key-value pair. It returns the level  $lv$ , the tree index  $tid_x$ , the slot index  $sid_x$  of the changed AMT, and the version  $ver$ .

---

```

1: procedure COMKV( $k, val$ )
2:   if KM contains  $k$ 
3:     ( $lv, sid_x$ )  $\leftarrow$  VM[ $k$ ];
4:   else
5:     ( $lv, sid_x$ )  $\leftarrow$  ALLOCATESLOT( $k$ );
6:     VM[ $k$ ]  $\leftarrow (lv, sid_x)$ ;
7:     ( $tid_x, lf$ )  $\leftarrow$  LEAFATLEVEL( $lv, k$ );
8:      $ver \leftarrow lf.ver[sid_x]$ ;
9:      $lf.ver[sid_x] \leftarrow lf.ver[sid_x] + 1$ ;
10:    Update the corresponding commitments and proofs.;
11:     $ver \leftarrow ver + 1$ ;
12:    return ( $lv, tid_x, sid_x, ver$ );
```

---



---

**Algorithm 3** A procedure to allocate a version slot to a new key. It takes the key  $k$  to allocate a slot for, and returns the level and the allocated slot index.

---

```

1: procedure ALLOCATESLOT( $k$ )
2:    $lv \leftarrow 0$ ;
3:   while true
4:     ( $tid_x, leaf$ )  $\leftarrow$  LEAFATLEVEL( $lv, k$ );
5:     for  $j \in [5]$ 
6:       if leaf.ver[sid_x] == 0
7:         leaf.keys[ $j$ ]  $\leftarrow k$ ;
8:         return ( $lv, j$ );
9:      $lv \leftarrow lv + 1$ ;
```

---

consist of the roots of both the top-level AMT and MPT.

The procedure COM first commits the key-value pairs in  $\mathbf{W}$  (Lines 3 to 6) with the sub-procedure COMKV. Then it updates the version numbers of all the affected sub-AMTs from the deepest sub-AMT to the root AMT (Lines 7 to 13) using the procedure UPDCOMVER. This procedure maintains



---

**Algorithm 4** A procedure to compute the AMT index and the leaf index of a key  $k$  at a AMT level  $lv$ . It returns the tree index  $tidx$  and the leaf  $leaf$  corresponding to the key  $k$  at level  $lv$ .

---

```

1: procedure LEAFATLEVEL( $lv, key$ )
2:    $tidx \leftarrow$  first bit to  $(k \cdot lv)$ -th bit of  $H(key)$ ;
3:    $lidx \leftarrow$   $(k \cdot lv + 1)$ -th bit to  $(k \cdot (lv + 1))$ -th bit of  $H(key)$ ;
4:    $leaf \leftarrow AM[(lv, tidx)].leaves[lidx]$ ;
5:   return ( $tidx, leaf$ );

```

---

**Algorithm 5** A procedure to update the commitment and version of an AMT at level  $lv$  and tree index  $tidx$ . It returns the commitment  $C$  and the updated version number  $ver$ .

---

```

1: procedure UPDCOMVER( $lv, tidx$ )
2:    $C \leftarrow AM[(lv, tidx)].comm$ ;
3:    $ptidx \leftarrow \lfloor tidx/n \rfloor$ ;
4:    $plidx \leftarrow tidx \bmod n$ ;
5:    $ver \leftarrow AM[(lv, ptidx)].leaves[plidx].ver[0]$ ;
6:   Increase  $AM[(lv, ptidx)].leaves[plidx].ver[0]$  by 1;
7:   Update the corresponding commitments and proofs;
8:    $ver \leftarrow ver + 1$ ;
9:   return ( $C, ver$ );

```

---

the version number for commitments of sub-AMTs similar to COMKV. While maintaining the version numbers, LVMT collects the tuples of keys, versions, values, and other metadata in a list  $\mathbb{M}$  (Line 5). The system treats a pair of the sub-AMT index and its commitment similarly to a key-value pair (Line 11). LVMT builds a Merkle tree for  $\mathbb{M}$ , thereby authenticating the value of a given key and version (Line 19). It also stores the positions of these elements in the Merkle trees (Lines 15 and 17). So when generating a proof, the prover can locate the corresponding Merkle leaves of a key or an AMT commitment.

The sub-procedure COMKV (Algorithm 2) is implemented to maintain and update the version numbers. COMKV( $k, val$ ) first finds the allocated version slot for the given key  $k$  (Line 3). If the key has not been allocated a version slot, it allocates a slot to it (Line 5). It uses the sub-procedure ALLOCATESLOT (Algorithm 3) to find a vacant slot in the AMT to allocate. In particular, starting from the root AMT, ALLOCATESLOT computes the tree and leaf indices for the given key at each level, checks if the leaf has a vacant slot, and then returns the level and slot indices of the slot; if the leaf doesn't have a free slot, it proceeds to the next level.

Then, COMKV computes the corresponding tree index  $tidx$  and the leaf  $lf$  for  $k$  at level  $lv$  (Line 7) using the sub-procedure LEAFATLEVEL (Algorithm 4), which finds the corresponding AMT index and leaf for the key  $k$  at the level  $lv$  using the hash  $H(k)$  of  $k$ . Since each AMT has  $m$  levels and  $2^m$  leaves, the first  $m \cdot lv$  bits of  $H(k)$  decides the AMT index and the subsequent  $m$  bits locate the leaf in the tree. Finally, COMKV locates the slot for this key and updates its version and other information according to AMT's rule (Line 8 to 10).

---

**Algorithm 6** A procedure to generate a proof for an existing key  $k$ . It returns the proof of the key.

---

```

1: procedure GENPROOF( $k$ )
2:    $keypf \leftarrow$  PROVEKEY( $k$ );
3:    $(lv, sidx) \leftarrow VM[k]$ ;
4:   while  $lv > 0$ 
5:      $tidx \leftarrow$  first bit to  $(k \cdot lv)$ -th bit of  $H(k)$ ;
6:      $commpfs[lv] \leftarrow$  PROVECOM( $lv, tidx$ );
7:      $lv \leftarrow lv - 1$ ;
8:   return ( $keypf, commpfs$ );

```

---

**Algorithm 7** A procedure to verify the proofs  $keypf$  and  $commpfs$  with respect to an AMT root  $aroot$  and Merkle root  $mroot$ .

---

```

1: procedure VERIFYPROOF( $keypf, commpfs, aroot, mroot$ )
2:   Verify the AMT proofs and the merkle proofs in  $keypf$  and  $commpfs$ ;
3:   Verify the commitment in  $commpfs[1]$  equals to  $aroot$ 
4:   if all the verification pass
5:     return true;
6:   else
7:     return false;

```

---

The sub-procedure UPDCOMVER (Algorithm 5) updates the commitment and its version number given an AMT located by its level and index.

## 4.2 Proving Key-value Pairs

As an authenticated storage, LVMT provides the following two interfaces to allow a user to query a value from an untrusted server and to verify the value with the commitment.

- GenProof( $k$ )  $\rightarrow \pi$ : Generates proof  $\pi$  for key  $k$ ;
- Verify( $k, v, \pi, comm$ )  $\rightarrow$  true/false: Verifies the key value pair  $(k, v)$  with respect to a ledger state commitment.

When responding to a query  $k$  from a light node, a full node will generate proof  $\pi$  using the procedure PROVE and responde with the loaded value and the current commitment.

The procedure PROVE (Algorithm 6) consists of two parts: 1) the proof of the value  $val$  of the key  $k$  with respect to the sub-AMT it belongs to (line 2) using the sub-procedure PROVEKEY; 2) the proof of the commitment for all the sub-AMT along the path from  $k$ 's sub-AMT to the root AMT (excluding the root AMT) (line 4 to line 7) using the sub-procedure PROVECOM.

The generated proof consists of a merkle proof for the existence of the tuple of the key (or the AMT index), the value (or the AMT commitment) and the version, an AMT proof for the version number, and other metadatas. We provide the definition for the sub-procedures PROVEKEY (Algorithm 8) and PROVECOM (Algorithm 9) in the supplementary material. In the supplementary material, we also discuss how to generate a non-existing proof.

The light node verifies the proof using the procedure VERIFY (Algorithm 7), which recovers the tuple of Merkle leaves to be verified from the proof and verifies the AMT proofs and the merkle proofs.

## 5 Implementation

We implemented the AMT using Arkworks [21], a Rust library for elliptic curve operations. AMT is built using the pairing parameters BN254 and supports vector commitment in the length of  $2^{16}$ . Each entry contains 254 available bits and is divided into six slots with 40 bits. For the public parameters required by the KZG commitment, we utilize the output from the Perpetual-Powers-of-Tau ceremony [27], which conducts an MPC protocol among over 70 participants worldwide in generating secure parameters. Based on the above AMT implementation, we implemented LVMT in Rust [1]. LVMT is compatible with any backend database that provides a key-value interface as defined in rust crate “kvdb” [39].

We ported the implementation of MPT from the OpenEthereum client [48], the most popular high-performance Rust implementation of Ethereum. We also implemented a variant of RainBlock [40], which developed an efficient MPT for distributed in-memory systems, by referencing its implementation [6]. This variant incorporates significant RainBlock features, including caching of top layers in memory, in-memory construction of the Merkle tree using pointers, and the application of lazy hash resolution. Unlike RainBlock, our variant stores the bottom layers on local storage instead of a distributed in-memory system. These implementation are also compatible with the same interface.

For the implementation of LVMT, we applied several optimizations:

**Combining entries in different maps:** For a given key, we use three maps KM, VM, and LM to store its value, version slot index, and the position of the Merkle tree for the recent change, respectively. In our implementation, we combine these entries into a single key-value pair to save read and write operation for each key.

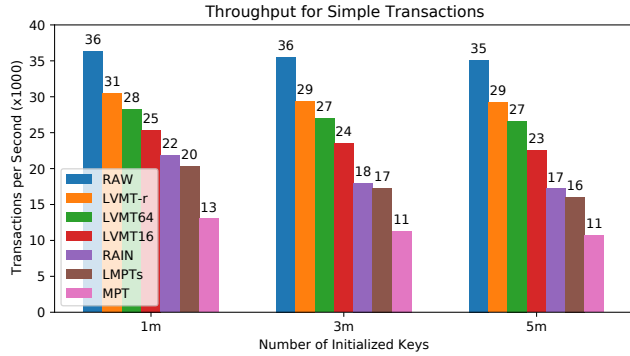
**Cache the root AMT:** The root AMT is frequently accessed. So its leaves and inner nodes of are always stored in memory. The commitments of the AMTs in the second levels are also cached. Each leaf and inner node of an AMT has two points on the elliptic curve. Given that we set the AMT height as 16, the root AMT and the commitments of AMTs in the second level store about 200,000 elliptic curve points in memory. Each point takes 96 bytes in our parameter, so roughly 20 MB of memory is needed to store them.

**Cache cryptographic parameters:** We expedited the com-

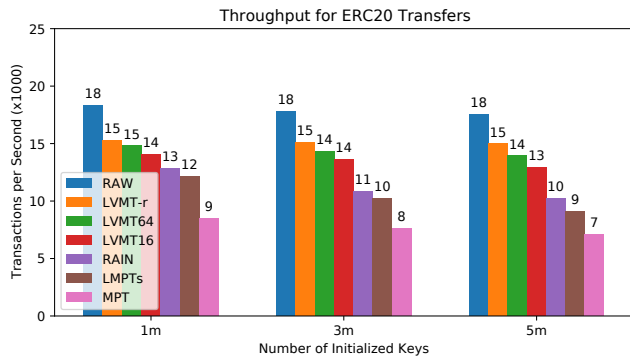
mitment update procedure by precomputing certain elliptic curve points. For instance, when the input entry at position  $i$  increases by  $\delta$ , the commitment can be updated as  $C' = C + P_i$ , where  $P_i = I_{i,n}(\tau) \cdot \delta \cdot G_1$ . Given that each entry is divided into six 40-bit slots, when the version number increases, the difference between the new and the previous version will be one of the following:  $1, 2^{40}, 2^{80}, 2^{120}, 2^{160}, 2^{200}$ . Thus, LVMT precomputes  $P_i^{(j)} = 2^{40j} \cdot P_i$  for all  $0 \leq j \leq 5$  and  $1 \leq i \leq n$ . So LVMT can simplify the commitment update procedure by merely incrementing a precomputed point. In our design, each elliptic curve point requires 96 bytes of storage. So a node excluding proof maintenance necessitates around 37 MB of memory. However, a node maintaining a shard of proof must cache additional parameters, resulting in a higher memory requirement, approximately 650 MB.

**Reduce the coordinates conversion time:** An elliptic curve point is uniquely represented by its affine coordinate  $(x, y) \in \mathbb{Z}_q^2$ , where  $q$  is a large prime number. These points can also be represented through projective coordinates  $(x, y, z) \in \mathbb{Z}_q^3$ , which accelerate arithmetic operations by eliminating division operations within a large prime field. The conversion of these projective coordinates back to the corresponding affine coordinates is given as  $(x/z^2, y/z^3) \in \mathbb{Z}_q^2$ . However, a challenge arises from the fact that a single elliptic curve point corresponds to multiple projective coordinates, leading to hashing inconsistencies. To address this issue, LVMT always converts the projective coordinates back to the affine coordinates when computing the hash of a sub-AMT commitment. This conversion process, however, is computationally intensive, taking approximately 60  $\mu$ s per conversion and can substantially impact the write speed. To alleviate this, we applied batch conversion of all projective coordinates to affine coordinates at the culmination of each block execution, decreasing the average conversion time to a mere 0.4  $\mu$ s.

**Garbage collection of append-only Merkle trees:** As a key’s version number increases, the old version tuples within the append-only Merkle trees become unnecessary for future proofs. When a subtree in a Merkle tree only has obsolete children, the entire subtree can be truncated, and only the subtree root is stored. A background thread performs this garbage collection to prevent impacting LVMT’s performance under heavy workloads. In a scenario where the append-only Merkle trees have accumulated  $m$  version tuples in the past, and only  $n$  tuples are currently active, the overhead of storing truncated Merkle trees is about  $(\log_2(m/n) + 1) \cdot 2n$ . (See appendix for the details.) While this introduces some additional overhead, it remains a practical approach.



(a) Transaction execution for balance transfers.



(b) Transaction execution for ERC-20 transfers.

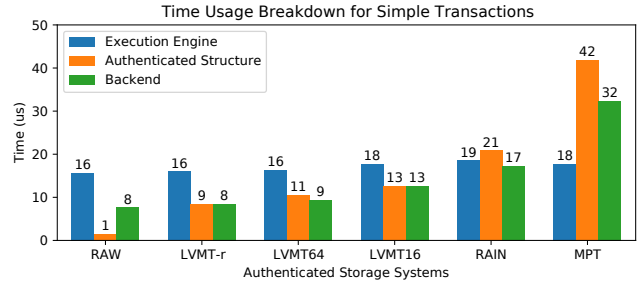
Figure 2: Throughput of transaction execution

## 6 Evaluation

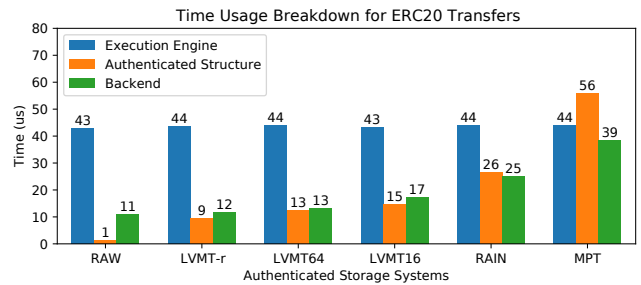
We evaluate LVMT’s performance and compare it to other authenticated storage systems using a machine with an Intel i9-10900K CPU, 32 GB DDR4 RAM, and SSD storages. All authenticated storage systems utilize RocksDB [47] as their backend key-value database.

We assess LVMT under different settings: 1) LVMT without associated information (no proof shard), 2) LVMT with 1/64 and 1/16 of the associated information (proof sharding), 3) LVMT with all associated information. In this context, LVMT-r represents LVMT without any associated information, while LVMT64, LVMT16, and LVMT1 signify LVMT with 1/64, 1/16, and complete proof sharding, respectively.

In addition to LVMT, we evaluate various authenticated storage systems for comparison. As previously mentioned, we have ported the MPT in OpenEthereum and have implemented a variant of RainBlock, which we refer to as MPT and RAIN, respectively. We also examine the Layered Merkle Patricia Tries (LMPTs) [20] utilized in Conflux [2, 33], a high-performance blockchain, represented by LMPTs. For reference, we also examine the performance of directly storing data into the backend, bypassing authenticated storage, denoted as RAW.



(a) Time cost breakdown for balance transfers.



(b) Time cost breakdown for ERC20 transfers.

Figure 3: Break down of the time usage in transaction execution on 5 million receivers.

**End-to-end performance:** We assess the end-to-end performance of authenticated storage on Conflux [2, 33], a high-performance blockchain. To gauge peak performance, we set a large block size of 20,000 transactions per block. Thus, all authenticated storage systems can finish executing one block within 0.5 to 5 seconds, aligning with the block generation intervals of major high-performance blockchains. To emulate the prevalent configuration of contemporary blockchains, we employ cgroup to restrict the memory consumption of a blockchain node to 16GB and allocate a 4GB RocksDB cache size. In the experiment, 10,000 senders randomly select addresses from the receiver space and transfer non-zero balances to them, representing simple payment transactions. We evaluate receiver spaces with one million, three million, and five million addresses. Conflux is run for an extended period, ensuring the number of executed transactions is three times larger than the receiver space.

Figure 2a shows that LVMT-r achieves a maximum throughput of 29669 TPS on average and is up to 2.7 times faster than MPT and 1.7 times faster than RAIN. We also evaluate the performance of transactions executing the transfer function of the popular ERC-20 smart contract [41], the most common transactions on the Ethereum blockchain [4]. As shown in Figure 2b, LVMT-r is up to 2.1 times faster than MPT and 1.5 times faster than LMPT in this workload.

To further study the time usage in execution of one transaction, we breakdown the time usage into three parts: 1) Execution Engine, i.e., transactions execution without access to the

authenticated storage, 2) Authenticated Structure, i.e., access to the authenticated storage without accesses the backend database, 3) Backend Database, i.e., accesses to the backend database. Figure 3a shows the breakdown of time usage in executing random balance transfer transactions with 5 million receivers. The execution engine takes the same time 16 us across the different storages. LVMT-r takes a similar time 11 us with RAW in accessing the backend. It implies LVMT-r almost eliminates the overhead of the authenticated storage from backend access. LVMT64 and LVMT16 take a similar time to LVMT. MPT requires 42 us and 33 us to access the authenticated structure and the backend database, respectively, which is more than 4x the time used in LVMT-r. As shown in Table 1, a single elliptic curve multiplication requires 92 us, which is even slower than MPT. Therefore, eliminating the expensive elliptic curve operation is necessary to make LVMT practical. Figure 3b shows the breakdown in executing random ERC20 transfers. The execution engine still takes the same time across the different storages but takes more time than the execution of the balance transfer. This is because the execution of ERC20 transfers requires more I/O accesses (e.g., loading contract bytecode). All the storages take about 20% more time than executing balance transfers.

This experiment shows that LVMT is able to maintain better throughput than MPT for both simple payment transactions and the typical ERC-20 smart contract transfer transactions.

**Stand-alone performance:** We also evaluate the stand-alone performance of authenticated storage systems in micro-benchmarks. We developed an authenticated storage benchmark tools [1] for evaluation. Since most transactions in the real world simply read the accounts of the sender and the receiver and update their balances, we launch a workload of 20 million random “read then write” operations and commit the changes every 100,000 operations, resembling a block being generated every several seconds. The authenticated storage is initiated with random key-value pairs whose size ranges from  $10^6$  to  $10^8$ . Both the key and the value are 256-bit strings. We use “1m”, “10m”, and “100m” to indicate the initialized size  $10^6$ ,  $10^7$  and  $10^8$ . Since LVMT needs to allocate version number slots for new keys, we also evaluate with a “fresh” setting: the storage has no initialization, and the workload accesses distinct keys.

In addition, to evaluate the performance under the real world access pattern, we extract the I/O trace on Ethereum, the largest smart contract platform. We choose transactions in 2021 winter, when Ethereum is going through its latest boom. We replay the Ethereum transactions from block 13,500,000 to block 13,600,000 to recover the I/O operations. These blocks access 22 million distinct keys, and make 97 million reads and 54 million writes in total. Each block contains an av-

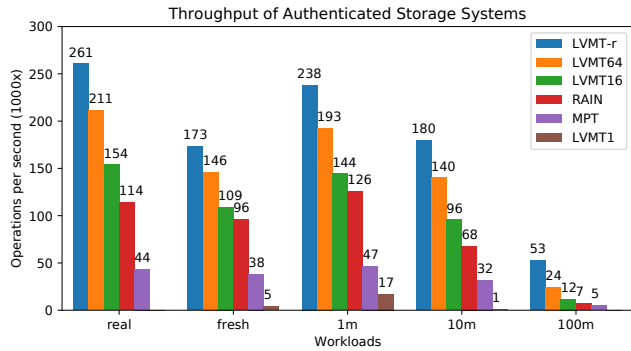
erage of 1,500 operations. Considering that high-performance authenticated storage can process over 100,000 operations per second, having only 1,500 operations per block results in an unreasonably short block generation cycle. This considerably impacts RAIN’s optimization efforts for lazy hash resolutions. To address this issue, we aggregated operations from every 50 blocks into a single block, making the block size in the real trace workload more closely resemble the size in a random access workload. We use “real” to denote the workload from real world transactions.

The primary blockchain node like Geth recommends a minimum of 16GB RAM for optimal performance. We assume that half of this memory is allocated for executing smart contracts that access authenticated storage systems, while the remaining half accommodates other functionalities. Consequently, we limited the runtime memory to 8GB using cgroups in our micro-benchmarks. We observed that authenticated storage systems without inherent caching strategies, such as RAW and MPT, perform better when provided with a higher RocksDB memory budget. Conversely, authenticated storage systems incorporating caching strategies, like LVMT and RAIN, show improved performance at a lower memory budget due to the need for an adequate filesystem cache. To optimize performance, we allocated a 4GB RocksDB cache size for RAW and MPT and a 2GB cache size for LVMT and RAIN. As the implementation of LMPTs is highly coupled with the backend database, and the vague boundary separating the authentication structure from the backend database posed a challenge to accurately gauge LMPTs in the micro-benchmarks. We removed LMPTs in micro-benchmarks.

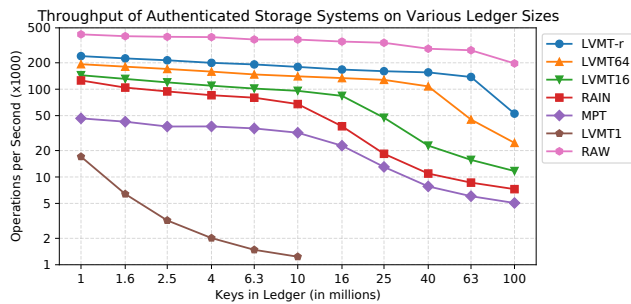
Figure 4a shows the throughput across various workloads. LVMT-r outperforms MPT and RAIN by at least 353% and 80%, respectively. When handling a shard of auxiliary information, LVMT64 and LVMT16 achieve roughly 80% and 60% of LVMT-r’s throughput across most workloads. LVMT1 consistently exhibits the weakest performance in all workloads, demonstrating the necessity of proof sharding. Within the Ethereum real trace workload, the ledger size initially comprises 4 million keys and eventually grows to 22 million keys. However, all the authenticated storage systems either outperform or match their performance in the ‘1m’ workload, as the real trace workload provides better access locality than random access.

Figure 4b illustrates the throughput for various ledger sizes. All authenticated storage systems experience a noticeable performance drop when reaching a specific ledger size threshold. This occurs because the ledger size surpasses memory limitations, preventing both RocksDB’s cache and the file system cache from effectively storing ledger data. RAIN and MPT performance begins to drop at a ledger size of 16 million,





(a) Throughput under different workloads.

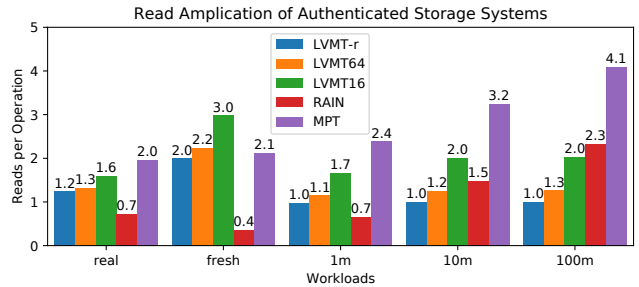


(b) Throughput for various ledger sizes.

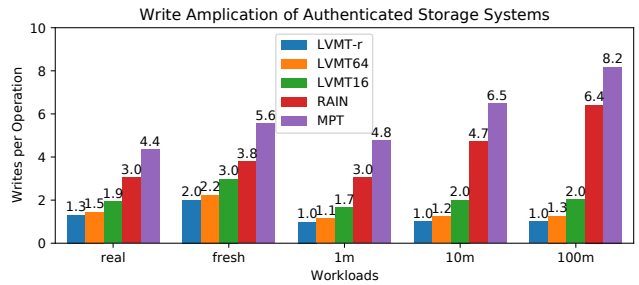
Figure 4: Throughput of authenticated storage systems.

whereas LVMT declines at a larger size. LVMT16, LVMT64, and LVMT-r demonstrate performance degradation starting from ledger sizes of 25 million, 63 million, and 100 million, respectively. This suggests that LVMT can provide efficient ledger access with a smaller memory usage.

**Read and write amplification:** We further study the read and write amplification at the backend database interface. Here, read amplification represents the ratio of backend read operations to those on authenticated storage systems' interfaces, and write operations are defined similarly. Figure 5a shows the read amplification under the different settings. As the ledger size grows, LVMT-r exhibits consistent read amplifications. The root AMT contains  $2^{16}$  entries, and the second level of AMTs  $2^{32}$  input entries in total. Since each entry has five slots for key-value pairs, the root AMT can only store 0.3 million keys, and the second level of AMTs accommodate 21 billion keys. So LVMT-r always requires two levels of AMT in all these workloads. The read amplification of a key grows linearly with its level in the AMTs, so it is reasonable for LVMT-r to exhibit similar read amplifications. In contrast, the read amplification of MPT grows from 2.4 to 4.1. RAIN demonstrates a smaller read amplification in the Ethereum real trace, indicating that its cache strategy benefits from better access locality in the real trace. For LVMT with proof shards, the read amplification grows linear with the size of auxiliary information. LVMT16 maintains four times the auxiliary information than LVMT64. So the surplus of LVMT16



(a) Read amplification of authenticated storage systems.



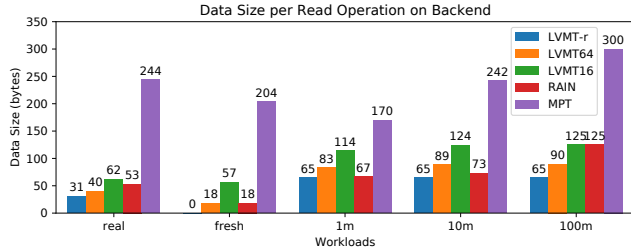
(b) Write amplification of authenticated storage systems.

Figure 5: Read and write amplifications of authenticated storage systems.

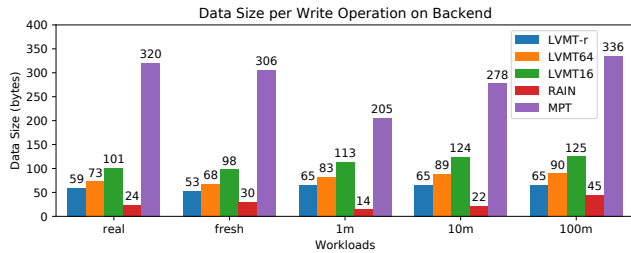
compared to LVMT-r is four times larger than the surplus of LVMT64. When accessing the fresh ledger state, allocating slots for the version number increases the read amplification of LVMT-r by 1.

Figure 5b displays the write amplification. The write amplification of LVMT is similar to the read amplification. MPT and RAIN have a larger write amplification than read amplification since MPT nodes are keyed by their hash digests. So each time the storage changes, a write operation and a deletion operation are applied to the backend.

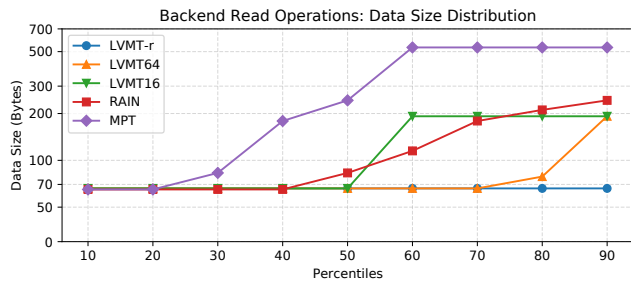
Figure 6a and 6b present the average sizes of read and write operations on backend, while figure 6c and 6d provides a more in-depth analysis of data size percentiles for the "100m" workload. Considering that each MPT node can accommodate up to 16 children, each containing a 32-byte hash, an MPT node may store around 500 bytes. So MPT's performance is negatively impacted by the combination of extensive read amplification and large data size per read operation. By caching the top six layers of MPT in memory, RAIN effectively reduces data sizes for both read and write operations. In RAIN, the first layer on disk represents the seventh layer of MPT, which can house roughly 17 million nodes. Thus, at the largest ledger size in our experiment, which consists of 100 million keys, each node only needs to accommodate six children, leading to a 200-byte node. LVMT-r only accesses elliptic curve points, which are 65 bytes in size. LVMT with proof shards may load 65-byte elliptic curve points and 192-bytes auxiliary information for an AMT node from backend.



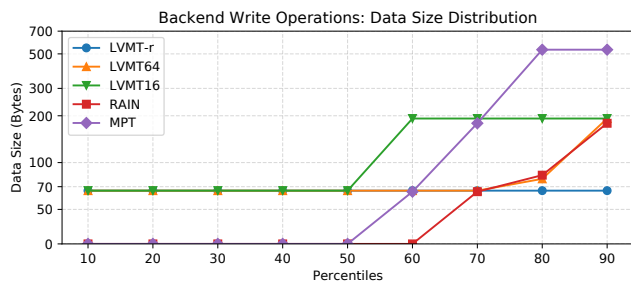
(a) Data size per read operation



(b) Data size per write operation



(c) Data size distribution of backend read operations



(d) Data size distribution of backend write operations

Figure 6: Data size of backend operations

Figure 6 indicates that around 40% of read operations for LVMT16 involve auxiliary information, while about 10% of LVMT64's read operations relate to auxiliary information.

## 7 Related Works

**Improved MPT structures:** mLSM proposes to maintain multiple levels of MPTs [43]. The most recent updates are in the lowest level (level 0). The key-value pairs in a lower level will be merged to higher levels periodically. LMPTs proposes maintaining three MPTs, one large MPT containing old state

and two small MPTs containing recent state changes [20]. LMPTs periodically merges small MPTs into large ones. For both mLSM and LMPTs, the concatenation of the Merkle roots of all the MPTs becomes the commitment for the ledger state.

Both LVMT and mLSM employ multi-level structures to minimize write amplification, but their approaches differ. mLSM maintains shallow top-level trees by regularly merging entries from the top-level Merkle trees into lower levels. Since write operations in mLSM only affect the top-level trees, the reduced depth decreases overall costs. LMPTs adopt a similar strategy, keeping a shallow delta MPT and periodically merging it into the snapshot. Conversely, LVMT's multi-level structure is akin to a tree, where each AMT serves as a node. When a write operation modifies an element in a lower-level AMT, all AMTs on the path from the root to the target AMT must be updated. With each AMT capable of accommodating up to 65,536 children, the high degree effectively reduces LVMT's overall depth, thus lowering write amplification.

Their techniques reduce the number of disk I/O operations on the critical path because the recently accessed state will be stored into MPTs with smaller depth, and the merge of MPTs can happen in a background thread. In contrast, LVMT almost eliminate unnecessary read amplification in practice. Our results show that when integrated end-to-end into Conflux, LVMT outperforms LMPTs by up to 2.5x. The mLSM paper only contains its conceptual design without implementation and evaluation [43]. It is unclear how mLSM would perform end-to-end with a blockchain in practice.

**Parallelize storage I/O:** RainBlock [40] introduces three different nodes in a blockchain system to accelerate the transaction execution: the storage prefetchers, the miners executing transactions, and the storage nodes. When executing transactions, the miners obtain needed data from multiple prefetchers and send the updates to multiple storage nodes. Each storage node maintains a shard of MPTs in memory. RainBlock changes the local storage I/O to network distributed storage I/O and benefits from the parallel I/O and in-memory storage. To reduce the read latency of network storage, RainBlock introduces I/O prefetchers and requires the miners to attach all the accessed key-value pairs and the witnesses (MPT nodes) when broadcasting blocks. RainBlock reports the average size of witnesses per transaction is 4 KB and their optimizations reduce the size of witnesses by 95%, so the additional network message per transaction is about 200 bytes, two times of a transaction. However, the inefficient usage of networks brings a bottleneck to a high-performance blockchain system [26]. RainBlock also suffers attacks in data availability. Since in-memory storage is costly, the number of replicas in RainBlock is much less than in Ethereum. As a comparison,

LVMT does not introduce additional network bandwidth consumption and data availability risk. Even if proof of shard in LVMT is lost, the other nodes can recover the auxiliary information of an AMT in minutes.

Both RainBlock and LVMT employ the sharding concept, but with different targets. RainBlock divides the ledger into multiple shards, preventing single nodes from accessing the entire ledger. To address this, RainBlock devised complex protocols between the prefetchers handling ledger reads and the miners executing transactions. Conversely, LVMT utilizes sharding solely to maintain auxiliary information for generating proof, allowing nodes to access the full ledger data during transaction execution. The proof sharding is mainly handled by blockchain node API providers. When users query a key, providers must direct the query to the corresponding node along with the relevant proof.

Another similarity between our RainBlock implementation and LVMT is caching top-level nodes. By default, RainBlock caches six layers of MPT nodes, while LVMT caches a single layer of AMT. As LVMT's nodes having a significantly higher degree than MPT (65,536 vs. 16), LVMT can use less memory to accommodate more entries in the first layer beneath the cached nodes.

**Vector commitment for data sharding:** Several vector commitment protocols [19, 24, 28, 30, 46, 49] have been proposed to reduce the proof size, support revealing elements in batch, or make the commitment efficiently updatable under some requirements. Some research also considers utilizing the vector commitment for data sharding on blockchain. Alin et al. [49] use KZG commitment protocol [28] to replace the underlying Merkle tree for data sharding. Unlike LVMT, the goal of this technique is not to improve the throughput but to reduce the data size of the blockchain storage. It requires the clients to maintain the proofs for their own data, keep updating the proof, and attach the values and proofs for the accessed storage in a transaction. Each client needs to be online and update the proofs of all of its data each time a write operation happens on the blockchain. Note that this protocol takes  $O(n)$  time to generate proof or maintain proofs for all data, which costs  $O(n)$  time to add a new key-value pair. It is therefore not designed for a high throughput blockchain system. When thousands of transactions are executed on the blockchain per second, a client cannot maintain its proofs efficiently.

Pointproofs [24] proposes an aggregatable and maintainable vector commitment protocol that can maintain the auxiliary information for proofs in  $O(\log n)$  time (like AMT) and reveal any  $k$ -element subset of elements in  $O(k)$  time with a batched proof. Pointproofs allows a consensus node to generate a batched proof for all the accessed key value pairs during block execution, so a node without the whole ledger

can verify the correctness of execution. However, for every 1024 transactions, Pointproofs takes 5 seconds to maintain the auxiliary information for proofs, which cannot match the requirements in a high throughput blockchain system.

**Accumulators:** Accumulators are cryptographic primitives that commit a set of elements to a short digest (commitment) while supporting operations like addition, deletion, membership proof, and non-membership proof. Merkle trees are one example of accumulators. A recent study [13] designed an RSA accumulator that supports batch operations and stores UTXO sets for a blockchain, with commitments updated in constant time.

In a zk-rollup blockchain [7], it is crucial to convince a light client with a SNARK proof [12] that the ledger root is updated correctly in a given sequence of operations. Ozdemir et al. replaced Merkle trees with RSA accumulators to accelerate SNARK proof generation [38]. Although RSA accumulators require  $O(n)$  time to generate a proof or update proofs for all elements, the time savings in SNARK proof generation outweigh the time spent in accumulator proof generation. However, in a high-performance authenticated storage, operations are processed in microseconds, rendering proof updates that require milliseconds per operation as relatively costly.

## 8 Conclusion

LVMT significantly reduces the disk I/O amplifications associated with each blockchain state access. When integrated into a high performance blockchain, LVMT has up to 2.7x higher throughput than the standard MPT structure. The promising results of LVMT demonstrate the potential of eliminating the performance bottleneck at the storage layer with vector commitment schemes.

## Acknowledgements

We express our gratitude to Peilun Li for his detailed guidance on the Conflux test framework, facilitating our end-to-end evaluations. We also appreciate the insightful suggestions from our shepherd, Micheal Wei, and the anonymous reviewers from EuroSys, S&P, and OSDI. Their critique and suggestions considerably improved our evaluation design and enriched our protocol discussion. This research has received support from the Shanghai Committee of Science and Technology, China (Grant No. 21511104600, 20DZ2221800), National Natural Science Foundation of China (Grant No. U2268202), and a gift fund from Nanjing Turing AI Institute.

## References

- [1] Authenticated storage benchmarks. <https://github.com/ChenxingLi/authenticated-storage-benchmarks>.
- [2] Conflux rust for authenticated storage benchmarks. <https://github.com/Conflux-Chain/conflux-rust/tree/asb-e2e>.
- [3] DefiLlama - DeFi Dashboard. <https://defillama.com>.
- [4] ERC-20 Top tokens. <https://etherscan.io/tokens>.
- [5] Patricia Tree. <https://eth.wiki/en/fundamentals/patricia-tree>.
- [6] Rainblock protocol. <https://github.com/RainBlock/rainblock-protocol>.
- [7] Zero-knowledge rollups. <https://ethereum.org/en/developers/docs/scaling/zk-rollups/>.
- [8] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.
- [9] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Proceedings of the 2002 International conference on security in communication networks*, pages 257–267. Springer, 2002.
- [10] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 2005 International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.
- [11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, pages 781–796, 2014.
- [12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, 2012.
- [13] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Proceedings of the 2019 Annual International Cryptology Conference*, pages 561–586. Springer, 2019.
- [14] Sean Bowe. BLS12-381: New zk-snark elliptic curve construction. <https://z.cash/blog/new-snark-curve>.
- [15] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Proceedings of the 2018 International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.
- [16] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *Cryptology ePrint Archive*, 2017.
- [17] Vitalik Buterin. Ethereum whitepaper. <https://ethereum.org/en/whitepaper/>.
- [18] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.
- [19] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Proceedings of the 2013 International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.
- [20] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. LMPTs: Eliminating storage bottlenecks for processing blockchain transactions. In *Proceedings of the 2022 International Conference on Blockchain and Cryptocurrency*. IEEE, 2022.
- [21] Arkworks contributors. arkworks zksnark ecosystem. <https://arkworks.rs>, 2022.
- [22] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, pages 45–59, 2016.
- [23] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.
- [24] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for



- multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2023, 2020.
- [25] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *Proceedings of the 2018 Annual International Cryptology Conference*, pages 698–728. Springer, 2018.
- [26] Yilin Han, Chenxing Li, Peilun Li, Ming Wu, Dong Zhou, and Fan Long. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 238–252, New York, NY, USA, 2020. Association for Computing Machinery.
- [27] Koh Wei Jie. Perpetual Powers of Tau. <https://github.com/weijiekoh/perpetualpowersoftau>.
- [28] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, pages 177–194. Springer, 2010.
- [29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE, 2018.
- [30] Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 2019.
- [31] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Proceedings of the 2015 International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.
- [32] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 438–453. ACM, 2020.
- [33] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Wei Xu, Fan Long, and Andrew Yao. A decentralized blockchain with high throughput and fast confirmation. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX, 2020.
- [34] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.
- [35] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.
- [36] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. <http://bitcoin.org/bitcoin.pdf>.
- [37] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erelay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 817–831, 2019.
- [38] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092, 2020.
- [39] Parity Technologies. Crate kvdb. <https://docs.rs/kvdb/0.4.0/kvdb/>.
- [40] Soujanya Ponnappalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347, 2021.
- [41] Ethereum Improvement Proposals. Eip-20: Token standard. <https://eips.ethereum.org/EIPS/eip-20>, 2015.
- [42] Pandian Raju, Soujanya Ponnappalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mlsm: Making authenticated storage faster in ethereum. In Ashvin Goel and Nisha Talagala, editors, *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX Association, 2018.
- [43] Pandian Raju, Soujanya Ponnappalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making authenticated storage faster in ethereum. In *Proceedings of the 10th USENIX*

*Workshop on Hot Topics in Storage and File Systems*, page 10, 2018.

- [44] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. Phantom and ghostdag: A scalable generalization of nakamoto consensus. *Cryptology ePrint Archive 2018/104*, 2018.
- [45] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Proceedings of the 2015 International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.
- [46] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, 2022.
- [47] Facebook Database Engineering Team. Rocksdb: A persistent key-value store for flash and ram storage. <https://rocksdb.org>, 2022.
- [48] Parity Technologies. Openethereum. <https://www.parity.io/ethereum/>, 2019.
- [49] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *Proceedings of the 2020 International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.
- [50] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 877–893. IEEE, 2020.
- [51] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 95–112, 2019.
- [52] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. OHIE: Blockchain scaling made simple. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 90–105. IEEE, 2020.
- [53] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

## Appendix

### Formal definition for inner nodes of AMT

We now provide formal definitions for the two elements associated with AMT inner nodes: a polynomial commitment and a batch proof about this polynomial function.

Since the auxiliary information is a binary tree, each node can be located by its depth and index. For a node indexed by  $i$  at depth  $d$ , its left and right children are assigned indices  $i$  and  $i + 2^d$ , respectively. The root is indexed by 0.

Let  $w$  be an  $n$ -th root of unity, such that  $w^n = 1$ , where  $n = 2^k$  for some integer  $k$ . Given an input  $\vec{a}$ , AMT constructs the vector commitment to  $\vec{a}$  to the polynomial commitment to  $f(x) : \mathbb{F}_p \rightarrow \mathbb{F}_p$  that satisfies  $f(w^i) = a_i$ , where  $\mathbb{F}_p$  is a prime field with order  $p$ . It is required that  $2^k | p - 1$ .

The interpolation for points is applied on roots in the  $\{i \in [n] \mid w^i\}$ , instead of  $[n]$ . This yields a Lagrange function which supports faster algorithms. The Lagrange function is defined as:

$$I_{i,n}(x) = \frac{\prod_{j \in [n] \wedge j \neq i} (x - w^j)}{\prod_{j \in [n] \wedge j \neq i} (w^i - w^j)},$$

where the numerator can be simplified to

$$\prod_{j \in [n] \wedge j \neq i} (x - w^j) = \frac{x^n - 1}{x - w^i} = \sum_{j=0}^{n-1} (x/w^i)^j,$$

and the denominator can be simplified to

$$\prod_{j \in [n] \wedge j \neq i} (w^i - w^j) = \sum_{j=0}^{n-1} (w^i/w^i)^j = n.$$

Thus,  $f(x)$  is built via Lagrange interpolation as:

$$I_{i,n}(x) = \frac{1}{n} \cdot \frac{x^n - 1}{x - w^i} \tag{2}$$

$$= \frac{\sum_{j=0}^{n-1} (x/w^i)^j}{n}. \tag{3}$$

So  $f(x)$  can be constructed by Lagrange interpolation as

$$f(x) = \sum_{i=1}^n a_i \cdot I_{i,n}(x).$$

Now we consider a node at depth  $d$  and index  $t$ , its associate function  $f_{d,t}(x)$  only mirrors  $f(x)$  at  $x = w^i$  where index  $i$  satisfies  $i \equiv t \pmod{2^d}$ , and then covers only the Lagrange interpolation terms of these elements:

$$f_{d,t}(x) := \sum_{i \in T_{d,t}} a_i \cdot I_{i,n}(x), \tag{4}$$

where  $T_{d,t} := \{i \in [n] \mid i \equiv t \pmod{2^d}\}$ . This node is associated with the commitment of function  $f_{d,t}(x)$  and the batch proof demonstrating  $f_{d,t}(w^i) = 0$  holds for all  $i \in [n] \setminus T_{d,t}$ . According to the KZG commitment, the commitment for  $f_{d,t}(x)$  is  $f_{d,t}(\tau) \cdot G_1$ , and the batch proof is  $h_{d,t}(\tau) \cdot G_1$ , where  $h_{d,t}(x)$  is defined by

$$h_{d,t}(x) := \frac{f_{d,t}(x)}{\prod_{i \in [n] \setminus T_{d,t}} (x - w^i)}, \quad (5)$$

with the denominator further simplifying to

$$\prod_{i \in [n] \setminus T_{d,t}} (x - w^i) = \frac{\prod_{i \in [n]} (x - w^i)}{\prod_{i \in T_{d,t}} (x - w^i)} = \frac{x^n - 1}{\prod_{i \in T_{d,t}} (x - w^i)}, \quad (6)$$

and where the denominator simplifies to

$$\prod_{i \in T_{d,t}} (x - w^i) = \prod_{i=0}^{2^{k-d}-1} \left( x - w^t \cdot \left( w^{2^d} \right)^i \right) = x^{2^{k-d}} - w^t \cdot 2^{k-d}. \quad (7)$$

For a leaf in subtree of this node with index  $s$ . If  $a_s$  increases by 1,  $f_{d,t}(x)$  and  $h_{d,t}(x)$  will be updated accordingly. By equation 4,  $f_{d,t}(x)$  will increase by  $I_{s,n}(x)$ , denoted as  $\tilde{f}_s(x)$ . By equation 5,  $h_{d,t}(x)$  will increase by  $\tilde{h}_{s,d}(x)$ , defined as:

$$\tilde{h}_{s,d}(x) := \frac{\tilde{f}_s(x)}{\prod_{i \in [n] \setminus T_{d,t}} (x - w^i)},$$

which can be simplified by equation 2, 6 and 7:

$$\begin{aligned} \tilde{h}_{s,d}(x) &= I_{s,n}(x) \cdot \frac{x^{2^{k-d}} - w^{s \cdot 2^{k-d}}}{x^n - 1} \\ &= \frac{1}{n} \cdot \frac{x^{2^{k-d}} - w^{s \cdot 2^{k-d}}}{x - w^s} \\ &= \frac{1}{n} \cdot \sum_{j=0}^{2^{k-d}-1} (w^s)^j \cdot x^{2^{k-d}-j}. \end{aligned}$$

In AMT, when increasing  $a_s$  by  $\delta$ , the commitments and proofs of the node along the path from the root to the corresponding leaf will increase by  $\delta \cdot \tilde{f}_s(\tau) \cdot G_1$  and  $\delta \cdot \tilde{h}_{d,s}(\tau) \cdot G_1$  respectively. The sequence of  $\{\tilde{f}_s(\tau) \cdot G_1\}_{s=1}^n$  and  $\{\tilde{h}_{d,s}(\tau) \cdot G_1\}_{s=1}^n$  for any  $d$  can be constructed by FFT. So the AMT can precompute  $O(n \log n)$  cached parameters in  $O(n \log^2 n)$  time and update the associated elements of each node with two multiplications and two additions on the elliptic curve.

## The overhead of storing the append-only Merkle trees

We provide a rough estimation of the overhead for storing truncated Merkle trees after garbage collection. Considering

**Algorithm 8** A procedure to prove a given key version. It returns the proof of the key version.

---

```

1: procedure PROVEKEY(k)
2:   (tidx, leaf) ← LEAFATLEVEL(lv, k);
3:   vers ← leaf.vers;
4:   C ← AM[(lv, tidx)].comm;
5:   (e, i) ← LM[k];
6:   val ← KM[k];
7:   (lv, sidx) ← VM[k];
8:   merklepf ← Prove the existence of (k, vers[sidx], val, lv, sidx)
   w.r.t. the current hroot
9:   amtpf ← Prove vers are the version numbers w.r.t. the commitment
   C
10:  return (merklepf, amtpf, vers, sidx, val, C);

```

---

**Algorithm 9** A procedure to prove the level lv and the tree index tidx of a sub-AMT. It returns the proof of the commitment of the sub-AMT.

---

```

1: procedure PROVECOM(lv, tidx)
2:   ptidx ← ⌊tidx/n⌋;
3:   plidx ← tidx mod n;
4:   vers ← AM[(lv - 1, ptidx)].leaves[plidx].vers;
5:   Cp ← AM[(lv - 1, ptidx)].comm;
6:   C ← AM[(lv, tidx)].comm;
7:   (e, i) ← LM[(lv - 1, ptidx)];
8:   merklepf ← Prove the existence of (lv, tidx, vers[0], C) w.r.t. the
   current hroot
9:   amtpf ← Prove vers are the version numbers w.r.t. the commitment
   Cp
10:  return (merklepf, amtpf, vers, Cp);

```

---

the roots of Merkle trees are organized in a tree, we can treat them as one large Merkle tree. We assume a full binary Merkle tree has  $k$  levels of inner nodes, accumulated  $m = 2^k$  version tuples, with  $n$  tuples currently active, where  $2^l \leq n < 2^{l+1}$  for some integer  $l$ . A node is not truncated if either itself or its sibling has active descendants, so each active tuple corresponds to at most two nodes per level. The bottom  $k - l - 1$  layers have at most  $2n \cdot (k - l - 1)$  nodes, less than  $2n \cdot \log_2(m/n)$ . The first  $l + 1$  levels have  $2^{l+1} - 1$  nodes, less than  $2n$ . Therefore, the maximum node count is  $(\log_2(m/n) + 1) \cdot 2n$ .

## Non-existence proof of LVMT

The process for generating a non-existence proof in LVMT is depicted in Algorithm 10. This procedure proves the non-existence of a key  $k$  by demonstrating that all potential version number slots for the key are already allocated to other keys.

It first allocate a version slot for  $k$  and followed by an immediate rollback of the allocation (lines 2-3). This process finds the next vacant slot for  $k$ .

Then, it proves the version number of this slot is zero, a process similar to Algorithm 6 except that it omits the merkle proof of the key (lines 5-12). This demonstrates that the slot is indeed unoccupied.

Last, it shows that all other potential slots for  $k$  are already allocated to different keys. It generates proof for them; the second fields of these proofs can be omitted since they have the same information as `commpfs` computed in line 11.

Thus, a non-existence proof in LVMT proves the absence of a key by showing that all its potential slots are occupied by other keys.

---

**Algorithm 10** A procedure to compute the non-existence proof for a given key.

---

```

1: procedure NONEXISTENCEPROOF( $k$ )
2:   ( $lv, sidx$ )  $\leftarrow$  ALLOCATESLOT( $k$ );
3:   Roll back the changes in allocating slot for  $k$ 
4:   ( $tidx, leaf$ )  $\leftarrow$  LEAFATLEVEL( $lv, k$ );
5:    $vers \leftarrow leaf.vers$ ;
6:    $C \leftarrow AM[(lv, tidx)].comm$ ;
7:    $amtpf \leftarrow$  Prove  $vers$  are the version numbers w.r.t. the commitment
    $C$ 
8:    $zeropf \leftarrow (amtpf, vers, sidx, C)$ ;
9:   while  $lv > 0$ 
10:     $tidx \leftarrow$  first bit to  $(k \cdot lv)$ -th bit of  $H(k)$ ;
11:     $commpfs[lv] \leftarrow$  PROVECOM( $lv, tidx$ )
12:     $lv \leftarrow lv - 1$ ;
13:     $L \leftarrow []$ ;
14:    for  $i \in [sidx - 1]$ 
15:       $keypf \leftarrow$  the first component of  $prove(leaf.keys[i])$ ;
16:       $L \leftarrow (leaf.keys[i], keypf) \cup L$ ;
17:    while  $lv > 0$ 
18:       $lv \leftarrow lv - 1$ ;
19:      ( $tidx, leaf$ )  $\leftarrow$  LEAFATLEVEL( $lv, k$ );
20:      for  $i \in [5]$ 
21:         $keypf \leftarrow$  the first component of  $prove(leaf.keys[i])$ ;
22:         $L \leftarrow (leaf.keys[i], keypf) \cup L$ ;
23:       $keypfs \leftarrow L$ ;
24:    return ( $zeropf, commpfs, keypfs$ );

```

---







# Honeycomb: Secure and Efficient GPU Executions via Static Validation

Haohui Mai<sup>1,\*,</sup> Jiacheng Zhao<sup>1,4,7,†</sup> Hongren Zheng<sup>2</sup> Yiyang Zhao<sup>1,7</sup> Zibin Liu<sup>6</sup>  
Mingyu Gao<sup>2</sup> Cong Wang<sup>5</sup> Huimin Cui<sup>1,7</sup> Xiaobing Feng<sup>1,4,7</sup> Christos Kozyrakis<sup>1,3</sup>  
*SKLP, Institute of Computing Technology, CAS<sup>1</sup> PrivacyCore Inc.<sup>1</sup> IIIS, Tsinghua University<sup>2</sup>*  
*Stanford<sup>3</sup> Zhongguancun Laboratory<sup>4</sup> IDEA Shenzhen<sup>5</sup> BUCT<sup>6</sup> UCAS<sup>7</sup>*

## Abstract

Graphics Processing Units (GPUs) unlock emerging use cases like large language models and autonomous driving. They process a large amount of sensitive data, where security is of critical importance. GPU Trusted Execution Environments (TEEs) generally provide security to GPU applications with modest overheads. Recent proposals for GPU TEEs are promising, but many of them require hardware changes that have a long lead time to deploy in production environments.

This paper presents Honeycomb, a software-based, secure and efficient TEE for GPU applications. The key idea of Honeycomb is to leverage static analysis to validate the security of GPU applications at load time. Co-designing with the CPU TEE, as well as adding OS and driver support, Honeycomb is able to remove both the OS and the driver from the trusted computing base (TCB). Validation also ensures that all applications inside the system are secure, enabling a concise and secure approach to exchange data in plaintext via shared device memory on the GPU.

We have prototyped Honeycomb targeting the AMD RX6900XT GPU. Honeycomb is evaluated on five representative benchmarks and 23 applications in total, covering workloads of high performance computing, deep learning, and image processing. The results show that Honeycomb is both *practical* and *efficient* to secure real-world GPU applications. Validating applications to run on Honeycomb requires modest developer efforts. The TCB is  $18\times$  smaller than the Linux-based systems. Secure inter-process communication is up to  $529\times$  faster. Moreover, running large language model workloads like BERT and NanoGPT has  $\sim 2\%$  overheads.

## 1 Introduction

Innovations in hardware accelerators and deep neural networks continue to enable personalized experiences for our physical and digital presences, reshaping areas ranging from

smart homes [34], virtual reality [85], to personalized cancer medicines [22]. Offering such intimate experiences heavily relies on large amounts of valuable and sensitive user data, which requires high levels of security and privacy support on hardware accelerators such as GPUs.

Trusted Execution Environments (TEEs) [4] encapsulate applications into enclaves to enhance security. TEEs enforce strong isolation among enclaves and the untrusted host environments, so that applications inside the enclaves can process plaintext data securely at native speed. For each enclave, all traffic that crosses its boundaries is encrypted to maintain confidentiality and integrity. Recent prototypes [28, 44, 45, 83] realize GPU TEEs with modest overheads, via serializing secure access to the GPU [28], augmenting the GPU hardware [83], customizing the I/O bus [44], or leveraging the sharing capabilities in device drivers [45].

This paper explores an alternative approach – using static analysis to *validate* that mutually distrusted GPU applications are confined to their enclaves. Intuitively, a validator inspects the binary code of GPU kernel functions (GPU kernels for short) to show that all possible execution traces maintain the confidentiality and integrity of the system, therefore these applications can safely share the GPU. This approach offers three benefits. First, it can complement the hardware limitations of existing GPUs. For example, low-cost GPUs such as the VC4 used by Raspberry Pi allow arbitrary writes to memory due to the lack of corresponding MMUs [16]. A validator can detect insecure behaviors and thwart the attacks by running standard static analysis such as def-use analysis and range checks on the GPU kernels. Moreover, advanced static analysis [59] might mitigate new attacks [19, 21] much faster compared to deploying new hardware supports in production.

The second benefit is that it allows more efficient implementations of current GPU TEEs. Shifting the runtime checks to load time removes them from the critical paths. Moreover, validating that applications that always have disjoint contexts might save the TEE implementation from flushing architectural contexts, including TLBs and buffer queues during every context switch [28], thus improving performances.

\*Haohui Mai is also affiliated with Hengmoxing Technologies.

†Jiacheng Zhao is the corresponding author.

Finally, validating every application provides a system-wide security invariant asserting that all applications are “good citizens”. The security invariant enables secure and efficient communication among enclaves. Real-world applications such as autonomous driving [89] and video analytics [66, 70] process data in multiple-stage pipelines. Separating each stage of the pipeline into different enclaves and connecting them using Inter-Process Communication (IPC) not only increases modularity and robustness, but also enables assembling the pipelines using mutually distrusted components from multiple vendors [63, 74]. Current GPU TEEs focus on strengthening isolation, for example, enforcing exclusive ownerships of GPU device memory [83]. Therefore two mutually distrusted enclaves need to tunnel the data through an encrypted shared buffer on the host memory for IPC. The overheads are prohibitive for production applications. For example, the GPUs on an autonomous vehicle process up to 50 GB/s of uncompressed video streams to make timely driving decisions [69]. Copying 50 GB/s of data to the host already takes up 30~40% of the total memory bandwidth of a commodity, high-end AMD Zen3 server, let alone the overheads of encrypting/decrypting the data. The capability of exchanging plaintext data directly in GPU reduces the overheads drastically, thus enabling real-world applications to migrate towards a more modular and robust architecture.

This paper presents the design and implementation of Honeycomb, a software-based, secure and efficient TEE for GPU applications. Honeycomb runs multiple mutually distrusted applications on the same GPU, and facilitates efficient and secure data exchange between applications. It supports common GPU workloads from simulations of molecular dynamics to training and inference of neural networks. All these capabilities of Honeycomb are built upon the idea of using static analysis to confine the behaviors of GPU applications.

Honeycomb faces three challenges to realize the three benefits and to provide a complete, real-world solution for GPU TEEs. First, it must balance the trade-offs between the capabilities and the complexities of the validator. A validator equipped with theorem provers gains their power, but then Honeycomb must include the theorem provers in the TCB, which is complex (e.g., Z3 4.12.2 has ~525 K lines of code) and occasionally error-prone [77]. On the other hand, a naïve validator might be insufficient to validate common security checks at load time, requiring inserting extra runtime checks that sit squarely on the performance critical paths.

Second, Honeycomb must minimize the end-to-end TCB to provide high confidence in security. The software/hardware stack of GPU applications is quite complex. For example, the compiler toolchain and the driver for the AMD RX6900XT GPU each consist of two million lines of code. Defects and vulnerabilities in these components are inevitable [23, 25, 26], but they should not compromise the security of Honeycomb.

Finally, Honeycomb must provide system-level support for secure and efficient IPC. The aforementioned plaintext IPC

among GPU enclaves can only be securely implemented if the data copies are cautiously initialized by the Honeycomb system and from/to strictly protected memory regions.

Honeycomb addresses the above challenges with three key techniques. First, the validator of Honeycomb performs static analysis of GPU kernels directly on binaries. It decodes the instructions of the GPU kernels to reconstruct the control and data flows. It models the memory access patterns using scalar evolution [6] and polyhedral models [14]. Our evaluation shows that the approach is effective to validate that the majority of memory accesses in GPU kernels are safe, because real-world GPU kernels tend to be well-optimized, having highly regular control flow structures and memory access patterns. The few remaining cases can be handled by inserting runtime checks, whose latencies are also well tolerated by the GPU memory hierarchy (§5).

Second, Honeycomb leverages hardware isolation mechanisms, and uses security monitors [54, 79, 90] to validate interactions in the system, so that it can minimize the trust on the software/hardware stack. Honeycomb launches applications inside CPU TEEs powered by AMD SEV-SNP [4]. The validator directly parses the GPU binaries to remove the compiler toolchain from the TCB. To remove both the user-space and kernel-space GPU drivers from the TCB, Honeycomb uses two security monitors to intercept and regulate all traffic between the applications and the GPU: (1) a Secure VM Service Module (SVSM) [4] running inside the application enclave, which enforces security policies at the application level (e.g., the application only launches validated kernels), and (2) a security monitor running inside a sandboxing hypervisor of the GPU, which regulates the behaviors of the GPU driver (e.g., the driver should never map a private memory page into two applications). Additionally, Honeycomb secures the data transfer between the CPU and the GPU to protect the confidentiality and integrity of the data (§6).

For the final challenge, Honeycomb reserves dedicated regions of the virtual address space for secure IPC to exchange plaintext data. Particularly, Honeycomb divides the virtual address space of each application into four regions: protected, read-only, read-write, and private. The validator ensures that application GPU kernels can only modify the private region. Putting the metadata and the receiving buffers into the protected and read-only regions prevents user applications from tampering with the IPC, reducing IPC in Honeycomb to copying plaintext data within the device memory (§7).

We have ported five representative benchmark suites, including the SpecACCEL 1.2 benchmark suites [76], inference applications of the ResNet18 neural network model [37] and the BERT language model [29], an application that trains GPT language models [48], and an image processing application that performs Canny edge detection [20], i.e., 23 applications in total. We have evaluated them on a server equipped with two AMD EPYC 7443 24-core processors and an AMD RX6900XT GPU. The results are promising. The TCB is

18× smaller than the Linux-based systems. A concise validator is sufficient to statically verify the security of large parts of GPU applications. Validating inference workloads on neural networks like ResNet18 and BERT requires adding zero runtime checks into the GPU kernels. Large language model workloads like BERT and NanoGPT have ~2% runtime overheads. IPC in Honeycomb is up to 529× faster than exchanging data using an encrypted, shared buffer on the host.

This paper makes the following contributions:

- The use of static analysis on GPU kernels to confine the behaviors of GPU applications to improve security. Our evaluations on five representative benchmark suites show that the analysis is both practical and effective to determine whether real-world GPU kernels are safe at load time with minimal additional runtime checks.
- The design and the implementation of a lightweight, end-to-end secure execution environment for GPU applications based on static validation.
- An IPC primitive that enables secure and efficient communications between GPU applications. The co-design of static analysis and OS support leads to a highly concise implementation.

## 2 Background

To understand the design of Honeycomb, it is important to first review the architectures and the programming interfaces of GPUs, as well as the basic concepts of polyhedral analysis [14, 35] used in this paper.

*Architectures and programming interfaces of commodity GPUs.* Modern GPUs offer the single instruction, multiple thread (SIMT) programming model to the applications. To run a workload, an application submits a launch request to the command queue of the GPU. The request specifies the binary function (i.e., GPU kernel), its arguments, the number of threads, and optionally, the size of a user-controllable, on-die high-speed scratchpad (i.e., shared memory) to perform the workload. Threads are organized into grids and blocks uniformly. Each grid consists of the same number of blocks, and each block consists of the same number of threads. Each thread within the same block has its own vector registers but shares access to the shared memory. The programming model provides a conceptual view where each thread executes the same instruction based on the values of its own registers. To achieve parallelization, each thread loads the inputs into its own registers and computes the outputs in parallel. Figure 1 presents an example of filling a region of memory to a specific value under the SIMT model.

The hardware architecture of GPUs closely matches the SIMT model above. A typical GPU consists of thousands of processing elements (PE) that are grouped into a three-level

hierarchy. The lowest level is called a warp, consisting of 32 or 64 logical PEs executed in lock-step. The micro-architecture (e.g., AMD GCN) might introduce parallel scalar units to perform uniform computation within a warp, or pipeline the computations on physical PEs to hide execution latency. Warps are further grouped into Compute Units (CU). A CU consists of a pool of vector registers and shared memory. Finally, a single GPU packages multiple CUs on the same die.

The hardware scheduler multiplexes the hardware resources across applications. The minimal scheduling unit is a warp. It always schedules all warps of a block within the same CU, therefore all threads within a block divide the vector register pool and share the same allocated shared memory inside the CU. The scheduler continuously schedules all the blocks and grids until the execution is completed.

The GPU driver creates a virtual address space for each GPU application. It allocates buffers for arguments and command queues out of the Graphics Translation Table (GTT) memory from the host. The buffers are mapped into the virtual address space on the GPU, from which the GPU kernels read the arguments and the layouts of grids and blocks directly.

*AMD SEV-SNP.* AMD SEV-SNP [4] (Secure Encrypted Virtualization-Secure Nested Paging) offers enhanced security features at the hardware level for Virtual Machines (VMs) running on an untrusted cloud system hypervisor. Similar to other TEEs, SEV-SNP supports remote attestation as well as both data confidentiality and integrity guarantees for the application VMs against untrusted host hypervisors. A dedicated hardware engine in the memory controller encrypts data before sending them to the off-chip main memory. SEV-SNP also tracks the ownership of each physical page with a Reverse Map Table (RMP) so that only the owner can write to a memory region. It further validates the page mapping to prevent malicious remapping of a single page to multiple owners. In such ways, it is able to alleviate typical data corruption, replay, memory aliasing, and memory remapping attacks.

In addition, SEV-SNP enables tagging each physical page with Virtual Memory Privilege Levels (VMPLs). It is similar to Ring 0-3 in the x86 architecture but for TEE VMs. One use case of VMPL is to implement Secure VM Service Module (SVSM). SVSM runs at VMPL0 and the guest operating system runs at VMPL1. SVSM can intercept syscalls and memory operations and serve as a security monitor.

*Polyhedral model.* The polyhedral model has been widely used in automatic parallelization and optimization of GPU programs [8, 14, 92]. Conceptually it represents each memory access as an affine expression (i.e. a linear combination) over an ordered set of loop variables. Analyzing the effects of memory access, such as aliasing and ranges, reduces to solving inequalities of integer variables. The polyhedral model works well with GPU kernels because they implicitly loop over the grids and the blocks, and performant GPU kernels have regular memory access patterns.



More concretely, an iteration vector  $I = (i_0, i_1, \dots, i_n) \in \mathcal{D}^s$  records the values of loop variables  $i_0, \dots, i_n$  for an instruction  $s$ . The domain  $\mathcal{D}^s$  is called the iteration domain. Note that the iterator vector usually includes the grid index (`gid`) and the local thread index (`lid`) for instructions in GPU kernels. An access function  $\mathcal{A}^s$  (w.r.t. instruction  $s$ ) takes an iterator vector as input and outputs the actual memory address.

Note that when  $\mathcal{A}^s$  is an affine function and  $\mathcal{D}^s$  is an affine space, all loops in  $I$  have fixed steps. For simplicity, we denote an access function as a vector with each element representing the coefficients of the corresponding dimension of the iteration vector. The dot product of the access function and the iteration vector is the actual memory address. We also introduce an extra dimension which always has the value 1 at the end of the iteration vector so that the access function can represent constant offsets in a uniform way.

Figure 2 shows the access functions of the kernel in Figure 1, a kernel filling a range of memory with a value. Affine operations on the values directly translate to affine operations on the vector forms of the corresponding access functions (e.g.,  $\mathcal{A}^5 = \text{dim} \cdot \mathcal{A}^3 + \mathcal{A}^4$ ), provided that `dim` is a constant throughout the analysis. The GPU kernel actually loads `dim` from the memory, however. In this case, security invariants in Honeycomb ensure that the value `dim` remains constant throughout the executions so that the analysis remains valid.

### 3 Threat model

In this paper, we adopt a similar threat model to previous studies on secure execution environments for GPUs [44, 45, 83]. The adversary controls the entire software stack, including the compiler toolchains, the operating system, the hypervisor, and the device drivers. It also has physical access to the server hardware and may sniff the PCIe traffic. We assume that the host machine CPU features TEE capabilities such as AMD SEV-SNP or Intel TDX [43], and the GPU features a hardware random number generator or performance counters to collect entropy for cryptographic uses. We also assume that users have the specifications of the server hardware and how it is connected, such as which PCIe slot that the GPU is plugged in. Finally we assume that Honeycomb is able to establish a trusted MMIO path with the GPU. Our prototype uses AMD SEV-TIO [1] to establish it, but such a path can also be realized using other secure I/O buses [44, 65], or alternatively, equipping the server with tamper detection mechanisms [75] and establishing a trusted I/O path to the GPU using a hypervisor [94]. We defer the details to §8.

The adversary can launch applications in Honeycomb, alter the results of the compiler toolchains, and tamper with the physical memory of the server. Additionally, the adversary can tamper with the DMA buffers. However, we trust the device memory of the GPU, since modern GPUs usually integrate the device memory using 2.5D/3D silicon interposers inside the same package. We assume that the adversary cannot observe

or corrupt the data stored in it [83]. Supporting integrated GPUs is out of the scope of this paper.

Similar to previous GPU TEEs [44, 83], side-channel attacks [17, 40, 82, 86] on trusted hardware are out of the scope of this paper. Honeycomb relies on the rich set of orthogonal work to alleviate these problems [9, 80]. Availability and denial-of-service attacks are also out of scope.

Under this threat model, Honeycomb should ensure confidentiality and integrity for multiple mutually distrusted applications running on the same GPU. The adversary cannot tamper with the code, the data and the control flows of both the CPU and GPU parts of the applications.

## 4 Overview

Figure 1 describes the overall architecture of Honeycomb. Honeycomb offers unified TEEs that cover both the CPU and GPU parts of the application. Honeycomb starts an application inside an AMD SEV-SNP TEE VM. It first starts the Secure VM Service Module (SVSM) at VMPL0. The SVSM bootstraps the BIOS, the guest Linux kernel, and finally the user-space application at VMPL1. SVSM regulates all interactions between the applications and the GPU. Recall that in CPU TEEs data are stored as plaintext within the CPU package. They are only encrypted when leaving for the off-chip main memory. In Honeycomb data on the device memory are stored decrypted, and the SVSM encrypts them when they are sent to the host. The path of reading data is similar.

The application requests GTT memory from Honeycomb to interact with the GPU. A piece of GTT memory can serve as a staging buffer for memory copies, which is mapped into the user-level address space, or serve as backing buffers for command queues, which are only accessible by the SVSM. In both cases the SVSM inspects the access to regulate secure memory transfers between the GPU and the applications [83], and launches validated GPU kernels with proper parameters. Note that although the current implementation of Honeycomb is based on AMD SEV-SNP, our design is applicable to other VM TEEs such as Intel TDX.

Honeycomb isolates the GPU inside a sandbox VM. The security monitor (SM) inside the sandbox is a hypervisor running below the Linux kernel. The SM regulates all interactions between the driver and the GPU. It ensures that the GPU follows the expected initialization sequences, and keeps track of the ownerships of the device memory pages to prevent accidental sharing of device memory among applications.

To execute GPU kernels, an application first loads the GPU binary that contains the GPU kernels into the device memory. The validator in Honeycomb takes both the binary code of a GPU kernel and the accompanying preconditions as inputs. It validates that each memory instruction in the GPU kernel can only access certain regions of the virtual address space. Note that the actual target addresses sometimes cannot be determined until the application executes the kernel with

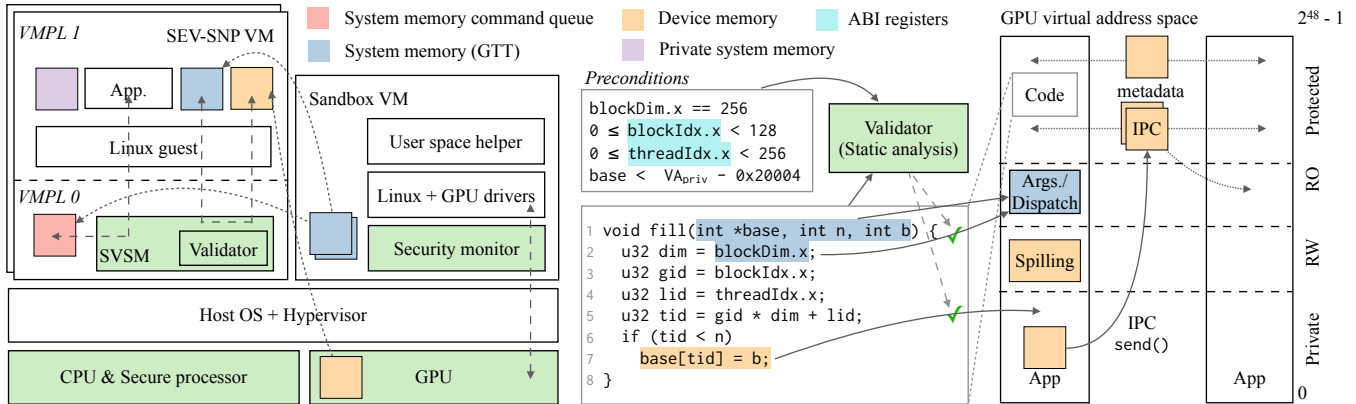


Figure 1: The overall architecture of Honeycomb. Left: Application VM and sandbox VM in their respective TEEs. Middle: Pre-conditions and source code of the GPU application kernel. Right: Layout of the virtual address spaces for two GPU applications. Long dashed arrows represent intercepted and validated requests by the validator (at load time) and the security monitors (at runtime). Dotted arrows represent physical memory page mappings.  $VA_{priv}$  is the topmost virtual address of the private region. Green boxes are components of the TCB.

$$\begin{aligned}
 \mathcal{D} &= \{(gid, lid) | 0 \leq gid < gridDim; 0 \leq lid < blockDim\} \\
 \mathcal{A}^3 &= (1, 0, 0) \\
 \mathcal{A}^4 &= (0, 1, 0) \\
 \mathcal{A}^5 &= dim \cdot \mathcal{A}^3 + \mathcal{A}^4 = (dim, 1, 0) \\
 \mathcal{A}^7 &= \mathcal{A}^5 + (0, 0, base) = (dim, 1, base)
 \end{aligned}$$

Figure 2: The iteration domain and the access functions (in the vector form) of the GPU kernel in Figure 1. The superscript denotes the corresponding statement. The parameter spaces of all access functions are  $(gid, lid, 1)$ .  $gridDim$  and  $blockDim$  describe the total number of grids and the number of threads in a block.

the concrete values of the arguments (e.g.,  $base$  in Figure 1). Therefore we introduce preconditions, which specify the constraints on the arguments so that the validator can analyze the bounds statically. Honeycomb checks the preconditions at runtime to ensure the attacker cannot subvert the analysis.

The validator decodes the instructions of the GPU kernel to reconstruct its control and data flows. It represents the target address of each memory instruction as a symbolic expression using scalar evolution and polyhedral models. It plugs in the preconditions to reason about the bounds of the target address, and ensures that the address stays within specified regions. The analysis is sound, meaning that once an access is proven, it is safe for all possible executions. For undecided cases like an indirect memory access  $a[b[i]]$ , Honeycomb requires the developer to annotate and add runtime checks to pass the validation. Our evaluation on real-world benchmark suites shows that the overheads of both development and runtime performance are modest – common production GPU kernels like matrix multiplications tend to have regular memory access patterns. The analysis is sufficient to capture the patterns, thus requiring few to none annotations.

The validator enforces access control that effectively divides the virtual address space of a GPU application into four regions: protected, read-only (RO), read-write (RW), and private, each of which has different access policies. For example, the application is prohibited to modify the RO region, but has full access to the private region. Honeycomb places the binary code and the arguments in the RO region so that a malicious kernel cannot modify the code on the fly after passing the validation. Furthermore, Honeycomb implements secure IPC through mapping the buffers into different regions. Honeycomb maps the IPC buffers into the sender’s protected and receiver’s RO region. The sender calls the trusted `send()` endpoint to copy the plaintext data to the IPC buffer, where both confidentiality and integrity are preserved.

## 5 Validator

The validator in Honeycomb checks the binary code for each GPU kernel of the application conforms with the following security invariants:

- *No dangling accesses.* A GPU kernel must never read uninitialized values from hardware registers.
- *All memory accesses reside in their regions.* All memory accesses to the memory regions conform with their access policies respectively.
- *Control flow integrity.* The execution must start at the designated entry point of the GPU kernel. The kernel can only transfer its control to the entry points of its basic blocks.

*Checking uninitialized uses of values.* The validator starts out parsing the binary code of the GPU kernel and building

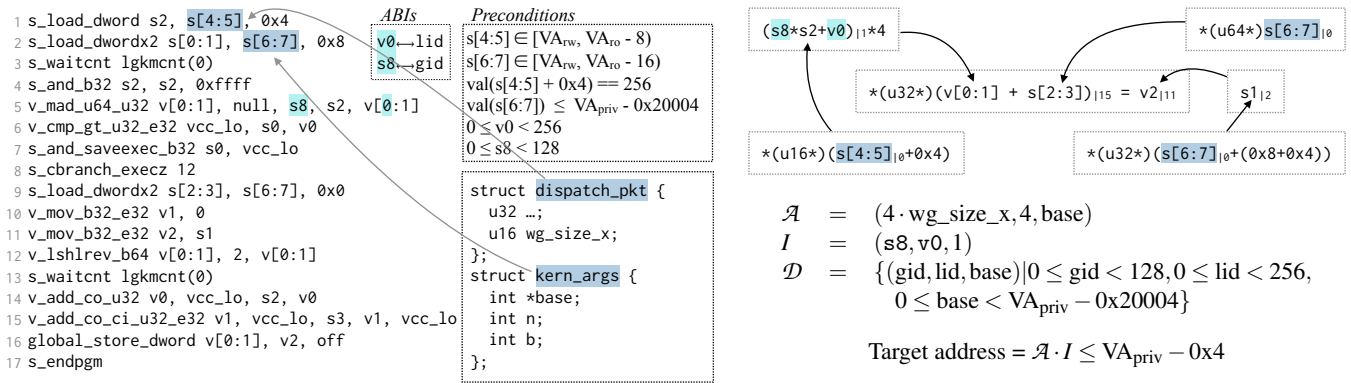


Figure 3: Workflow of validating the binary code generated from Figure 1. Left: the assembly, preconditions and the ABIs. Top right: the value chain of the symbolic expression that represents the target address of Line 16. Subscripts represent the line number on the left at which the value should be considered. Bottom right: the access function  $\mathcal{A}$ , the iteration vector  $I$  and the iteration domain  $\mathcal{D}$ .

the Static Single-Assignment (SSA) representation and the Control Flow Graph (CFG) for each kernel function. The validator checks dangling accesses by inspecting whether the SSA representation of the kernel function is valid.

*Checking memory accesses.* Figure 3 presents the overall workflow when validating the program described in Figure 1. For each memory instruction, the validator constructs a symbolic expression and derives the access function  $\mathcal{A}$  to represent the target address of each memory instruction. The algorithm combines scalar evolution analysis and polyhedral models, and is flow-sensitive and path-insensitive.

The validator further derives the iteration vector  $I$  and the iteration domain  $\mathcal{D}$  from the application binary interface (ABI) and the preconditions of the GPU kernel. Recall that the dot product  $\mathcal{A} \cdot I$  computes the value of the target address. It is sufficient to plug in  $\mathcal{D}$  to compute the range of the target address and to verify whether the memory access is inbound.

A closer look at Figure 3 shows that the validator must address practical complexities when analyzing the binary code. For example, the compiler promotes the load of `dispatch_pkt.wg_size_x` into a 32-bit load instruction (Lines 1 and 4). It also lowers a 64-bit addition into two instructions (Lines 14-15). The validator heuristically rediscovers their semantics when constructing the symbolic expressions. Additionally, the validator matches sequences of instructions to rediscover semantics of divisions, modulus, and min/max operators.

Another example is that conventional polyhedral models require all multipliers to be constants. The value of `s2` comes from a load instruction (Lines 1 and 4), breaking the subsequent analysis when constructing a polyhedral representation of the global ID (Line 5). The validator recognizes that the instruction is loading from the RO region and it is safe to treat it as a constant in the analysis. Such relaxation is essential to derive  $\mathcal{A}$  and eventually to validate that Line 16 is safe.

Aggressive compiler optimizations can create additional

burdens for analysis. For example, the definition and the usage of a value could be scattered in two basic blocks separated by other basic blocks in the CFG. They are guarded by the same condition so the program is valid at runtime but a path-insensitive algorithm fails to connect them. More powerful analysis or language-level support [30, 41, 60] will address the issue but we intentionally limit the capabilities of the validator to bound the size of TCB. Honeycomb requires the developer to alter the GPU kernel to pass the validation. Additionally, the validator requires the developer to add runtime checks for indirect memory accesses like `a[b[i]]` since it does not fully track the memory access of the heap.

We found that the simple algorithm is effective against commonly used production kernels such as matrix multiplications or element-wise transformations as the analysis perfectly captures the regular and predictable memory access patterns commonly seen in most GPU kernels.

*Enforcing control flow integrity.* It is relatively straightforward to decode GPU kernels since modern GPUs have RISC-style instruction sets. The validator simply validates that all branches jump to valid instructions. The validator does not support indirect branches. Although based on our experience they are rarely used in real-world GPU kernels, the developer can turn indirect branches to a series of branches that have explicit targets. The validator does not support self-modifying code to ensure the integrity of the analysis.

## 6 Security monitors

There are two types of security monitors in Honeycomb to regulate the interactions with the GPU. In Honeycomb every application runs inside its own TEE VM. The SVSM regulates the interactions between the application and the GPU. The security monitor (SM) in the sandbox VM regulates the interactions between the driver and the GPU. The SM also keeps track of the ownership of memory pages to prevent ac-

cidental sharing between applications. Together they are able to remove the OS kernel and the GPU driver from the TCB. Similar to existing GPU TEEs [83], Honeycomb implements the following functionalities.

*Initialization.* Honeycomb enforces the untrusted GPU driver to follow a correct sequence to initialize the GPU. However, to our best knowledge, no public specifications are available for our target device, the AMD RX6900XT GPU. We therefore collect the trace of an initialization sequence on the baseline platform and use it as the ground truth. We further inspect the source code of the driver to build state machines to model the initialization sequence. The SM intercepts all MMIO traffic to ensure that the GPU driver follows the transitions of the state machines. The SM directly passes the firmware to the GPU since the hardware will validate its integrity with cryptographic signatures.

Despite the fact that there is no specification, we were able to find five bugs in the AMDGPU driver that violate security. More specifically, there are two instances where the parameters of the hardware queues are initialized with incorrect values, two instances where the queues are prematurely enabled before all parameters are set, and one instance of out-of-bound access on the hardware buffer. All five bugs are confirmed by upstream developers, and their corresponding fixes have been deployed since Linux 5.19.

*Launching GPU kernels.* Applications call the same user-space APIs (e.g., the HIP APIs [2]) to launch GPU kernels on Honeycomb. First, applications call `hipModuleLoadData()` to load GPU binaries. The implementation of the API traps into the SVSM, where the SVSM validates the kernels, then copies the kernels into the protected region and records their preconditions given that they have passed the validations.

Applications call `hipLaunchKernel()` to launch a GPU kernel. Similarly, its implementation traps into the SVSM, where the SVSM confirms the preconditions are valid with respect to the actual arguments. It then updates the command queue to enqueue the launch if preconditions are satisfied.

*Isolating address spaces.* On the CPU side, Honeycomb leverages existing mechanisms in SEV-SNP TEE to enforce isolation between different applications. SEV-SNP ensures the integrity of VM data and protects against various vulnerabilities, including replay and remapping attacks (§2).

On the GPU side, the SM intercepts all traffic between the driver and the GPU to maintain a RMP table similar to Graviton [83] to track the ownership of the pages. The Linux driver allocates page tables inside the device memory and updates them through MMIO requests. The SM intercepts these requests and updates the RMP table. Additionally, the SM prevents applications from mapping the page tables into their address spaces to subvert the isolation.

*Securing data transfers.* Honeycomb implements secure data communication channels between the GPU and the host CPU, and coordinates all data transfers into and out of the GPU

device memory. All transfers between the host and the device memory must be done via a special trusted kernel in Honeycomb, with all transferred data encrypted and authenticated under an ephemeral encryption key. Honeycomb disallows the applications from mapping the host memory into their address spaces or directly creating DMA queues.

One practical issue is how to bootstrap and maintain the secure channel. Honeycomb uses the `s_memrealtime` instruction to get the value of the real-time counter on the AMD RX6900XT GPU. Honeycomb launches a kernel to perform reads, invalidating caches to generate entropy and extract them. The entropy is used to establish a shared security key using Diffie-Hellman key exchange [31]. Honeycomb stores the entropy in the protected region to prevent user applications from accessing it.

## 7 Secure and efficient IPC

Honeycomb enables two enclaves to securely exchange plaintext data within the device memory. To make an IPC, Honeycomb maps the IPC buffer to the sender's protected region and the receiver's RO region. The sender calls `send()` to initiate the IPC. `send()` is a trusted endpoint that simply copies the data into the protected region and updates the indices of the IPC buffers. The GPU kernels on the receiver side can read the RO region directly but need to update the indices via `recv()` provided by Honeycomb. The scheme is secure because no GPU kernels from the user applications can access the protected region nor write to the RO region.

For simplicity, the current prototype of Honeycomb maps all IPC buffers to the protected regions consistently across all applications so that it is possible to identify the endpoints using only the virtual addresses. Adding finer-grained access control through capabilities [33] is relatively straightforward.

To summarize, the ultimate simplicity comes from the guarantee that none of the user-provided GPU kernels inside Honeycomb is able to tamper with the data in the protected, RO and RW regions, making exchanging data between two enclave applications in Honeycomb as simple as copying a piece of memory.

## 8 Discussion

*Establishing a trusted I/O path to the GPU.* When the server does not have AMD SEV-TIO or other secure I/O buses, Honeycomb can leverage prior work [94] to establish a trusted I/O path to the GPU. On the high level, Honeycomb acquires exclusive control of the I/O paths at the beginning of the boot-up process, before any untrusted components can access the GPU. Particularly, the server first boots into the SM where the whole boot-up process is validated and attested via SecureBoot [55]. Second, the SM enumerates the PCIe buses to discover the MMIO regions of the GPU and all of its



upstream PCIe switches. Then it initializes the IOMMU to protect the MMIO regions from any unauthorized accesses by the hypervisor and other devices' DMA buffers. It further programs the PCIe Access Control Services (ACS) registers [5] to stop unauthorized peer-to-peer PCIe transactions. After these steps Honeycomb can continue the normal boot-up process. By correctly configuring the IOMMU and by disabling peer-to-peer PCIe accesses, the above initialization process is sufficient to isolate the MMIO regions of the GPU from the untrusted software components and other I/O devices inside the server [94]. Honeycomb relies on additional tamper detection mechanisms to detect and mitigate physical attacks.

In the alternative setup above, the TCB must include all components used in the boot-up process, such as the UEFI BIOS. The SM and the untrusted hypervisor must be adopted to support nested virtualization [11]. Legacy systems without tamper detection mechanisms have a weaker threat model as they are unable to defend against physical attacks.

*Remote attestations.* The user application needs to attest both its own SEV-SNP VM and the sandbox VM to validate the security of the execution environment. It follows the standard procedures to attest its own VM. The attestation also guarantees a valid execution context of the GPU since the SVSM is part of the attestation, and it regulates the GPU execution context. Note that the SM is part of the TCB. The SM maintains a public-private key pair for the attestation of the sandbox VM. The user application authenticates the SM using the key pair to validate the security of the sandbox VM.

## 9 Security analysis

*Attacking the software stack.* An attacker might try to launch a malicious GPU kernel by subverting the validation in Honeycomb. Some possible attacks include invalidating the preconditions with invalid arguments, subverting the data flows via manipulating the values of spilled registers, and modifying the code or the target addresses of a branch to hijack the control flows [18]. Recall that SEV-SNP ensures that the user application cannot tamper with the SVSM. The SVSM reevaluates the preconditions with the arguments before executing the GPU kernels to defend against the first attack. Honeycomb protects the code and the spilling regions in the RO and RW regions to defend against the second and third attacks. Particularly, the validator analyzes each global memory access in the GPU kernel to ensure that it cannot tamper with the reserved regions, maintaining the integrity of the validation.

An attacker might tamper with the system software stack, including the GPU driver, the host operating system, and the hypervisor, to subvert the security of Honeycomb. To gain access to the plaintext information residing in the device memory, they might execute code to craft malicious MMIO requests, to initiate unauthorized DMA requests, or to map the device memory from other applications to different address

spaces. This is ineffective because the SM in Honeycomb regulates all MMIO and DMA requests from the system software stack. By intercepting the MMIO requests, it enforces isolation on address spaces and prevents accidental sharing. It also enforces that data communication with the external world is all encrypted and authenticated.

The attacker might alter the GPU firmware or divert from the designated bootup sequences in order to control the GPU. This is ineffective because the GPU hardware verifies the integrity of the firmware [55], and the SM in Honeycomb validates the bootup sequences during GPU initialization.

*Attacking the hardware stack.* An attacker might interpose the host memory to try to alter the trusted components like the SVSM in Honeycomb. This is ineffective because SEV-SNP includes attestation procedures to verify the integrity of the trusted components. SEV-SNP also incorporates memory encryption and integrity to defend against the attack.

An attacker might interpose on the PCIe fabrics to insert MMIO or DMA requests, or tamper with existing requests to access the plaintext information residing in the device memory. Alternatively, they might map the MMIO regions of the GPU to another I/O device or initiate peer-to-peer PCIe transactions to interact with the GPU. Both types of attacks are ineffective when the GPU is attached to a secure I/O bus [1, 44, 65]. When using the alternative initialization process described in §8 to establish a trusted I/O path, Honeycomb detects and stops the first type of attacks using tamper detection mechanisms [75]. To defend against the second type of attacks, Honeycomb programs the IOMMU and PCIe ACS registers to acquire exclusive control on the MMIO regions of the GPU before starting any untrusted components. Additionally an attacker might write to the I/O ports that map to the registers in the PCIe configuration space, in the hope of relocating the MMIO regions of the GPU. Honeycomb is able to identify and stop potential attacks as the hardware topology uniquely determines the mappings [94]. An attacker might also initiate peer-to-peer PCIe transactions between an I/O device and the GPU bypassing the IOMMU. Honeycomb stops the attacks because it programs the PCIe ACS registers to prevent unauthorized peer-to-peer PCIe transactions.

Our threat model assumes that an attacker cannot snoop or tamper with the device memory of the discrete GPU. The attacker can also try to perform the row hammer attack [51], which can be mitigated by orthogonal research [7, 67, 87].

*Side-channel attacks.* An attacker might try to exploit various timing and power side channels. Defending them is out of the scope of this paper and can leverage orthogonal work [9, 80].

## 10 Implementation

We have implemented Honeycomb on top of Rust 1.64.0 nightly with about 32,000 lines of code. The current prototype supports the x64 architecture and the AMD RX6900XT GPU.

We use the AES256-GCM [32] to encrypt and decrypt traffic between the host and the GPU.

The validator understands the AMDGPU ELF binary format and disassembles the machine code of the GPU kernels of the AMD RDNA2 ISA. The structures of scalar evolution analysis and polyhedral representations closely resemble the corresponding parts in LLVM [53].

We have implemented both the SVSM and the SM in Rust. The SM is implemented as a Type I hypervisor. We have implemented the user-space runtime, including the corresponding bindings of HIP and OpenCL in C++, in around 8,500 lines of code. The user-space runtime is outside the TCB.

## 11 Evaluation

The evaluation of Honeycomb tries to answer the following questions both qualitatively and quantitatively:

- Does static validation in Honeycomb improve security?
- Is Honeycomb practical for real-world applications?
- Where do the overheads in Honeycomb come from?
- How efficient is the IPC in Honeycomb?
- How much effort is required to adopt Honeycomb for new applications?

### 11.1 Experiment setup

We evaluate Honeycomb on a server equipped with two 24-core 2.85 GHz AMD EPYC 7443 CPUs, 128 GB DDR4 memory, and a 480 GB SAMSUNG PM893 SSD. The server has an AMD RX6900XT GPU that has 16 GB of device memory. It connects to a gigabit Ethernet with the Broadcom BCM5720 Ethernet adapter. The machine runs a patched Linux 5.15.0 kernel to support SEV-SNP VMs. Both the sandbox and the application VM runs Linux 5.17.0 on top of QEMU 7.1.0. We have not yet enabled SEV-SNP for the sandbox VM due to complications of passing the AMD RX6900XT GPU directly into the VM. We use the ROCm 5.4.0 [3] GPU driver when running the baseline experiments. We pin all applications to the first CPU socket where the GPU is attached.

### 11.2 TCB

Honeycomb provides a secure and efficient execution environment for GPU applications. To quantitatively evaluate our efforts, we count the lines of code (LOC) in the TCB of both Honeycomb and the Linux platform using SCC [15]. The current prototype of Honeycomb only supports a limited set of hardware, thus we only count the lines of code for the x64 platform and the essential parts of the driver for AMD RX6900XT. Figure 4 presents the counts of LOC for the TCB of both Honeycomb and the Linux platform.

Honeycomb provides security guarantees with respect to the threat model in Section 3 with an order of magnitude smaller TCB compared to the normal Linux platform. The security of a GPU application running on Linux relies on the correctness of both the kernel space and the user space (ROCm) of the GPU driver. The result of the smaller TCB is consistent with other systems that adopt the design of security monitors [79, 90]. The SM and the validator in Honeycomb separate the concerns of enforcing security from implementing the required functionalities, removing the heavy-lifting portions (e.g., Linux) of the system out of the TCB.

| System                       | LOC         |
|------------------------------|-------------|
| <i>Honeycomb</i>             | 82,738      |
| SVSM                         | 9,839       |
| SM+Sandbox VM                | 9,376       |
| Validator                    | 12,299      |
| Rust runtime                 | 50,864      |
| <i>Linux 5.17</i>            | ~ 1,503,519 |
| Core functionalities for x86 | 844,993     |
| AMDGPU driver for AMD 6900XT | 607,689     |
| Kernel libraries (DRM & TTM) | 50,837      |
| ROCm 5.4.0                   | 397,151     |
| HIP Library                  | 188,995     |
| ROCR Runtime                 | 73,241      |
| ROCm Common Runtime          | 62,173      |
| ROCR Thunk interface         | 72,742      |

Figure 4: Estimated LOC for TCBs of Honeycomb and Linux. It also shows the LOC of some major components in the TCB.

### 11.3 End-to-end performance

We choose five representative benchmark suites to study how Honeycomb performs on real-world workloads:

*SpecACCEL*. SpecACCEL is a performance test suite that represents high-performance computing applications like simulations of computational fluid dynamics and molecular dynamics. We evaluate all 19 OpenCL applications in the SpecACCEL 1.2 benchmark suites. All benchmarks are evaluated against the default parameters and the reference input size.

*ResNet18*. ResNet18 is an 18-layer convolutional neural network model. It is a popular image classification model that is used on low-power edge devices. We implement a benchmark that classifies 10 images using the ResNet18 model. The model uses the single-precision, pre-trained weights (IMAGENET1K\_V1) from PyTorch 1.12.1 [64].

*BERT*. BERT is a large transformer model that powers various natural language processing tasks. We derive a benchmark from the NVIDIA FasterTransformer backend [62]. We use the BERT\_BASE configuration [29]. The model has 12 layers



Figure 5: Relative execution time for the five benchmark suites evaluated on Honeycomb.

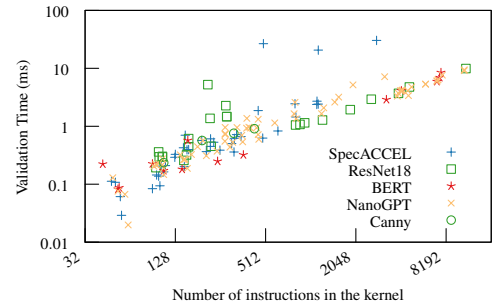


Figure 6: The time spent on validating GPU kernels in wall clock vs. the size of the kernels.

and 110M parameters, preloaded with the single-precision, pre-trained weights called bert-base-uncased [38]. The benchmark reports the time of performing a single shot of inference on BERT.

*NanoGPT.* NanoGPT is a minimal implementation of training medium-size Generative Pre-trained Transformer (GPT) models. GPT models are often used to power chat bots or to generate human-like content. We implement a benchmark that fine-tunes the GPT2 model [68] using the tiny Shakespeare dataset in the NanoGPT repository. We preload the weights of its 124M parameters from [39]. The benchmark trains with a batch size of 4 and uses  $\sim 15$  GB out of the 16 GB of total device memory available.

*Canny.* Canny implements the Canny edge detection algorithm to detect edges in images. We implement a benchmark that detects edges on an image in the UHDSR4K dataset [91]. The resolution of the image is  $3840 \times 2160$ .

Figure 5 presents the relative execution time of all five benchmark applications. The relative execution time ranges from 0.89 (104.lbm) to 1.27 (Canny). Large language models in Honeycomb are particularly efficient: the relative slowdowns of BERT and NanoGPT are 2% and 0%. This is because their execution time is dominated by matrix multiplications, whose memory accesses can be efficiently reasoned about with scalar evolution analysis and polyhedral models. The validator requires no runtime checks to be inserted into the performance-critical, general matrix multiplication (GEMM) GPU kernels to pass validation. Honeycomb essentially launches the exact same GPU kernels compared to the baseline.

Figure 5 further breaks down the overheads into four categories: (1) Driver (slowdowns from an alternative driver), (2) SVSM (validating the requests in the command queues), (3) Mem (securing memory transfers) and (4) V (runtime checks). The characteristics of runtime overheads vary among applications. First, the alternative driver is simpler and faster in general but lacks the optimizations on large memory copies. Running Canny on the alternative driver is 18% slower (7.16s

vs. 6.11s) because it loads an 8MB image into the GPU before processing it. Second, to enforce security the SVSM must inspect each request of kernel launch. The overhead is more evident for applications that mostly consist of small, fast GPU kernels like ResNet.

The third source of overheads is secure memory transfer. For example, 117.bfs copies the frontier and the tail of the BFS queue back and forth between the host and the device in each iteration, transferring 400 bytes of data for 108,000 times. Enabling secure memory transfer results in a 42% slowdown (13.97s vs. 9.82s). 116.histo also has significant overheads because it uses memcpy() to zero out a piece of device memory at the beginning of each iteration. Changing it to memset() eliminates the overheads.

The final source of overheads comes from the runtime checks in GPU kernels that are inserted to facilitate validations. Runtime checks slow down 121.lavamnd by 19% (5.80s vs. 4.87s). However, most of the overhead can be attributed to one single runtime check. The GPU kernel writes to `a[b[i] + threadIdx.x * j]` in two-level nested loops. `i` and `j` are loop variables thus `b[i]` remains constant in the outer loop. Developer must insert a runtime check to aid the validation since `b[i]` is a value from the memory where the validator does not model. Note that the runtime check can be hoisted to the outer loop since checking the indices at the first and the last iterations of `j` is sufficient to guarantee safety. Hoisting the check effectively eliminates the overheads (both 4.87s for disabling runtime checks and hoisting the check to the outer loop). The case of 128.heartwall is similar. Extending the validation to understand hoisting is left to future work.

## 11.4 Overheads

The previous subsection has discussed the overheads on the runtime checks inside the GPU kernels. This section further studies other overheads introduced by the system design of Honeycomb, namely:

- Validating the GPU kernels at load time.

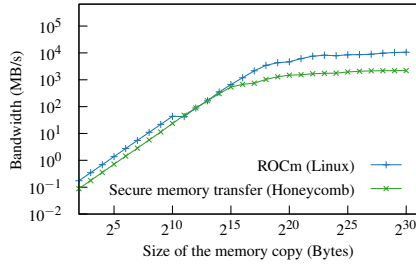


Figure 7: Round-trip bandwidth of data copies of various lengths between the host and the GPU, with and w/o secure memory transfers.

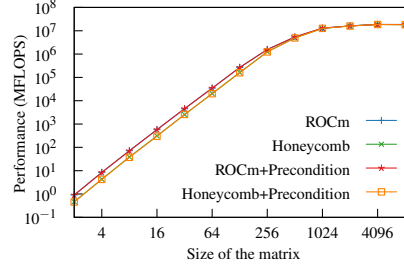


Figure 8: The achieved FLOPS in matrix multiplication on various sizes of the matrices. Both the x and y axes are on log scales.

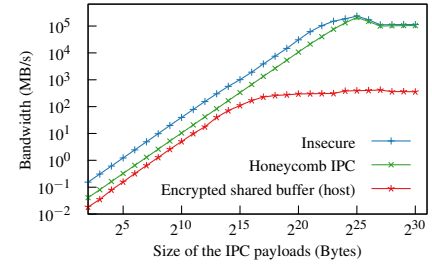


Figure 9: Round-trip IPC bandwidth of the ping-pong application with different sizes of payloads.

- Securing memory transfers between the host and the GPU.
- Checking the preconditions against the arguments and launching the GPU kernels.

*Overheads of validating GPU kernels.* Figure 6 describes the time spent on validating the GPU kernels in all five benchmark suites we evaluate vs. the number of instructions they have. The time used by validation is roughly linear with respect to the size of the GPU kernel. Out of 149 GPU kernels we have evaluated, the largest one is a GEMM GPU kernel that has 11297 instructions coming from rocBLAS [49]. The longest time spent on validating an individual GPU kernel is around 30 ms (128.heartwall). At the application level, validating NanoGPT training takes the longest time in our evaluation. It consists of 73 GPU kernels, taking 162 ms in total to validate all of them. Note that the validation is a one-time overhead when loading the applications. Real-world GPU applications like training execute the kernels continuously for days. The evaluation shows that validating GPU kernels is efficient and has negligible overheads on overall application performance.

*Overheads of secure memory transfers.* We study the overheads of secure memory transfers using a benchmark that transfers data back and forth between the host and the GPU. A round trip of a secure memory transfer includes (1) encrypting the data on the host CPU, (2) copying the encrypted data to and from the GPU, and (3) decrypting the data. We warm up the benchmark for 5 seconds and report the average transfer bandwidth over a 30-second period for various sizes of transfers. Figure 7 presents the round-trip data bandwidth with and without secure memory transfers. The bandwidths of both ROCm and Honeycomb first increase linearly with the sizes of the payloads and then peak at 10.63 and 2.20 GB/s. The bandwidth is bounded by AES encryption/decryption throughput of a single CPU core.

*Overheads of checking preconditions and launching GPU kernels.* We study the performance impacts of checking preconditions in Honeycomb by measuring the performance of

multiplying two single-precision square matrices of various sizes in Honeycomb. We implement the benchmark using the GEMM GPU kernels from rocBLAS. The validator has verified that all global memory accesses in these GPU kernels are safe, thus there are no extra runtime checks inside the GPU kernels. We have further ported all checks on preconditions directly into the benchmark, making precondition checking the sole overhead in this benchmark.

Figure 8 presents the achieved FLOPS on both Linux and Honeycomb against various sizes of the square matrices (from  $2 \times 2$  to  $8192 \times 8192$ ), with or without checking the preconditions. Honeycomb performs 41 range checks on the kernel arguments to validate the preconditions on each launch, taking roughly  $0.04\mu\text{s}$  to complete. All GEMM GPU kernels in the benchmark have the identical function signature. Both the number of checks on preconditions and the performance are consistent.

Honeycomb is slightly slower than Linux when the size of the matrices is less than 1024, because SVSM must check each request to ensure that applications can only launch validated kernels. We observe that the overhead is  $\sim 8\mu\text{s}$  per launch of GPU kernels. To cross-validate the overheads, we compare the latency of launching a no-op kernel on Linux and on Honeycomb. The average latencies over a million launches on Linux and on Honeycomb are  $13.15\mu\text{s}$  and  $25.06\mu\text{s}$ . The overhead of checking preconditions is two orders of magnitude smaller than launching a no-op kernel.

## 11.5 IPC performance

We study the case of exchanging data between two TEE applications on the same host. We compare the bandwidth between exchanging the data via (1) an encrypted shared buffer on the host, and (2) the IPC mechanism in Honeycomb, where no encryption is needed. We have built two applications and evaluated their performance: (1) a ping-pong application that sends data back and forth, and (2) a two-stage image processing application that mimics the perception pipelines in autonomous vehicles. It ingests a video stream in an enclave



and performs edge detection in another one. The isolation not only increases modularity and robustness, but also simplifies the integrations with third-party vendor SDKs.

Figure 9 presents the effective bandwidth of the ping-pong application with different sizes of IPC payloads, along with a reference, insecure implementation that copies the payloads within the device memory using `hipMemcpyDtoD()`. IPC in Honeycomb is  $2\text{-}529\times$  faster compared to exchanging data via a shared encrypted memory buffer on the host. The effective bandwidth of round-trip IPCs peaks at  $\sim 233$  GB/s (89% of the reference insecure implementation) when sending payloads of 32 MB, where three buffers of such size utilize the shared L3 cache of the GPU (128 MB). In contrast, the bandwidth of using an encrypted shared buffer for IPC peaks at  $\sim 411$  MB/s. We attribute the inefficiency to the fact that GPU TEEs make secure memory transfers transparent to applications. Other GPU TEEs cannot give out the encryption keys to the applications without compromising the security, so the only way of sharing data securely is to re-encrypt the data before sharing them, where the performance is bounded by the CPU performance of encryption and decryption as shown in Figure 7.

We have assembled the Canny application into a two-stage image processing pipeline. The end-to-end latencies of processing a single frame of 4K image are  $679\ \mu\text{s}$  and  $18579\ \mu\text{s}$  when using direct IPC and an encrypted shared buffer on the host, where  $617\ \mu\text{s}$  is spent on actual computation.

## 11.6 Developer experience

Figure 10 presents the metrics on the kernels and development efforts of the five benchmark suites we have evaluated. It presents the number of GPU kernels, the number of memory instructions, the number of runtime checks inserted, and the number of preconditions written for each application. Neural network applications ResNet18, BERT, and NanoGPT are considerably bigger, where the GEMM kernels contribute to more than 70% of the total number of instructions. RocBLAS launches different GEMM kernels based on the sizes of matrices for optimal performances.

*Experience with neural network models and the Canny edge detector.* We have ported 109 GPU kernels in total for ResNet18, BERT, Nano and Canny. We are able to classify the GPU kernels used in neural network models into three categories: (1) element-wise operations, (2) matrix operations, including multiplication, transposition and convolution, and (3) special-purpose GPU kernels such as Im2d2col or radix sort. Note that developers do not directly write the GPU kernels. The GPU kernels either come from well-optimized libraries such as MIOpen [49] or are generated by PyTorch.

We found that GPU kernels in the first two categories have well-optimized, regular memory access patterns. Scalar evolution analysis and polyhedral models are sufficient to verify the safety of the memory accesses, meaning that no extra runtime

checks are required. However, it is important to extend the polyhedral models to treat the values of some kernel arguments as constants (§5) to complete the analysis. Many of these GPU kernels are generic library functions. They take the shape and the length of the data as arguments, which are often used in calculating addresses. GPU kernels used in the Canny edge detector also fall within the first two categories.

GPU kernels in the third category require case-by-base discussion. The class of Im2d2col GPU kernels used by ResNet18 essentially unrolls a matrix into a long vector under different configurations. The challenge of analyzing their memory access is that the GPU kernels use division and modulo operations to transform the basis of indices. For example, the statements `out_x = inner_lid % out_cols_wg;` `out_y = inner_lid / out_cols_wg;` repartition the index `inner_lid` based on the value of `out_cols_wg`. It is easy to see such accesses are inbound but neither the standard scalar evolutions nor polyhedral representations can model them. Such repartitions are often parts of the tight loops thus extra runtime checks can incur significant performance overheads. Fortunately, we found out that the compiler generates pretty stable code sequences for these statements. We have implemented a pattern matching algorithm to iterate over the instructions to uncover the semantics of repartitions, so that the validator can verify these Im2d2col GPU kernels without the need of runtime checks.

Radix sorts are introduced to speed up the training of neural networks on GPUs. Particularly, during the backward propagation pass the training application sorts the sparse gradients before propagating the values in order to improve locality and to save the precious memory bandwidth. While radix sorts are efficient on GPUs, they pose challenges for validation due to the presence of indirect memory references. It is a non-goal for the validator in Honeycomb to verify the safety of indirect memory references, so we have added runtime checks to the sorting kernels in the NanoGPT training application. The overall overheads are insignificant as radix sorts are accountable for less than 0.02% of the total running time. Replacing radix sort with an algorithm like merge sort that is more friendly to validation may be a good alternative.

Many preconditions are mechanical and usually straightforward (e.g., ensuring that the whole matrix is in the private region). Since GPU kernels take data shapes as inputs, all of which must be specified in the preconditions. For instance, each GEMM kernel requires 30 preconditions. Writing these preconditions is tedious, and we have developed a script to generate the preconditions automatically.

In short, it requires inserting zero runtime checks into ResNet18, BERT and Canny to pass validation in Honeycomb. We introduce runtime checks in the NanoGPT training application with negligible performance overheads. Developing preconditions for the GPU kernels requires modest effort. Patching frameworks like PyTorch to use the validated versions of the GPU kernels, however, turns out to be a big-

| Benchmark    | 101.tpacf | 103.stencil | 104.lbm | 110.fft | 112.spmv | 114.mriq | 116.histo | 117.bfs | 118.cutcp | 120.kmeans | 121.lavamd | 122.cfd | 123.nw | 124.hotspot | 125.lud | 126.ge | 127.strad | 128.heartwall | 140.bplustree | ResNet18 | BERT  | NanoGPT | Canny |
|--------------|-----------|-------------|---------|---------|----------|----------|-----------|---------|-----------|------------|------------|---------|--------|-------------|---------|--------|-----------|---------------|---------------|----------|-------|---------|-------|
| Kernels      | 1         | 1           | 1       | 1       | 1        | 2        | 5         | 2       | 1         | 2          | 1          | 5       | 2      | 1           | 3       | 2      | 6         | 1             | 2             | 24       | 14    | 67      | 4     |
| Mem. instrs. | 15        | 15          | 30      | 9       | 9        | 21       | 77        | 14      | 10        | 14         | 7          | 119     | 76     | 8           | 105     | 20     | 60        | 121           | 46            | 1,531    | 1,768 | 7,202   | 55    |
| Checks       | 0         | 0           | 0       | 0       | 5        | 0        | 1         | 8       | 3         | 0          | 5          | 5       | 0      | 0           | 0       | 0      | 6         | 14            | 16            | 0        | 0     | 44      | 0     |
| Preconds.    | 5         | 8           | 5       | 6       | 9        | 13       | 29        | 16      | 8         | 12         | 4          | 29      | 20     | 8           | 9       | 13     | 45        | 25            | 19            | 443      | 181   | 1,529   | 18    |

Figure 10: Metrics on the kernels and development efforts to validate GPU kernels in the five evaluated benchmark suites.

ger practical challenge. We eventually end up patching the userspace runtime to load the validated GPU kernels.

*Experience with the SpecACCEL benchmark suites.* We have ported all 19 benchmarks (40 kernels in total) in the SpecACCEL 1.2 benchmark suites to Honeycomb. We classify the required changes into three categories: (1) adding optimization, (2) undoing optimization, and (3) indirect heap references.

*Adding optimization.* The validator can benefit from optimizing the GPU kernel. For example, 110.fft has a division instruction in the kernel. The divisor is a power-of-2 constant. Propagating it into the GPU kernel reduces the division into bit shifts, simplifying the validation.

*Undoing optimization.* Aggressive optimization in compilers issue instruction sequences that are difficult to model in scalar expression. For example, the compiler compiles the expression  $-1-bx$  in 123.nw to a single instruction `s_not_b32 bx`. It is difficult for the validator to model such an instruction as a scalar expression. We have to rewrite the expression to undo the optimization so that the validator can recognize the expression.

*Indirect heap references.* There are 9 benchmarks that have indirect heap references in the code. Each instance of irregular heap access requires adding a runtime check which incurs runtime overheads. For example, 118.cutfp casts a float to the index of an array; other benchmarks like 112.spmv expose patterns like `a[b[i]]`. All these instances require adding runtime checks to pass the validations.

## 12 Related work

*TEE designs on GPUs.* GPU TEEs enforce isolation among mutually distrusted enclaves. Graviton [83] augmented the GPU hardware with RMP tables to isolate physical memory pages among enclaves. Telekine [42] was built upon Graviton to remove a side channel regarding the execution time of GPU kernels, enhancing the overall isolation confidence. HIX [44] and CRONUS [45] relied on the GPU driver’s isolation mechanisms to properly protect and isolate applications. However, modern GPU drivers are inherently complex, and even security features like isolation are prone to vulnerabilities [24, 27]. StrongBox [28] leveraged the secure IOMMU on the SoC to

isolate enclaves on integrated GPUs. It required updating the IOMMU and flushing the IOMMU TLB to switch between different execution environments.

HETEE [95] deployed a cluster of tamper-resistant servers with commodity GPUs. These servers accessed secure accelerator boxes through a centralized FPGA-based controller, achieving isolation through physical separation. Visor [66] focused on privacy-preserving video analytics in the cloud. It combined oblivious algorithms at the application level and a hybrid TEE at the system level to provide isolation.

Honeycomb enforces isolation via confining the behaviors of GPU applications with static analysis. Honeycomb’s approach complements the hardware limitations of existing GPUs, reduces overheads, and creates opportunities for optimizations like directly sharing data via IPC. Performing IPC in current GPU TEEs requires copying the data back and forth through an encrypted shared memory buffer on the host. Honeycomb combines confinements from static analysis and system-level designs to reduce IPC into copying plaintext within the device memory. IPC in Honeycomb is up to two orders of magnitude faster than conventional methods, enabling real-world applications to adopt a more modular architecture with modest overheads.

GPU TEEs also enforce isolation between enclaves and the untrusted host environment. They need to establish secure communication channels between the application running inside the CPU TEE and the GPU. Prior work implemented end-to-end secure communication channels in the GPU hardware [83], in the PCIe fabrics [44], or leveraging the secure IOMMU inside the SoC [28, 45]. Honeycomb leverages existing work on secure I/O bus [1, 44, 65] or software-based solutions [94] to establish secure communication channels.

*Crypto-based secure computing on GPUs.* Recent advances in modern cryptography offer theoretically provable solutions for privacy-preserving computing, such as Multi-Party Computation (MPC), Garbled Circuit (GC) and Homomorphic Encryption (HE). These algorithms have been used to realize secure GPU computations for machine learning and data analytics. On top of GAZELLE [47], Delphi [56] used GPU to accelerate the HE-based linear operations, and also selectively replace the expensive GC-based nonlinear ReLU opera-

tors with polynomial approximations. CryptGPU [78] further implemented both linear and nonlinear operations in MPC-based protocols on GPUs. It embedded the secret-shared value computations into floating-point operations, effectively utilizing GPU hardware units. GForce [61] instead focused on inference and addressed the high latency of non-linear operators by applying new quantization approaches and employing GPU-friendly protocols. Finally, Piranha [84] was a general and modular framework for accelerating secret-sharing-based MPC protocols on GPUs, leveraging optimized integer-based GPU kernels and memory-efficient in-place computations.

The cryptographic solutions do not keep the plaintext values in the untrusted platforms, so they are more resistant to side-channel vulnerabilities. However, their substantially higher computational cost causes huge slowdown compared to native processing. Specialized hardware [50, 71, 72] and trusted hardware units [93] have been proposed to accelerate HE and MPC, but all require non-trivial hardware changes.

Slalom [81] took a different approach. It used a CPU TEE to compute the non-linear parts, and offloaded encrypted data to the untrusted GPU to process linear operations. Both confidentiality and integrity are guaranteed. DarKnight [36] further optimized the flow with a better encryption method that greatly reduced the communication cost between CPU and GPU as well as the computations involved in the CPU TEE.

*Secure operating systems.* There is fruitful research on improving the security of operating systems, including explicitizing the security policies [73, 90], applying safe languages in the OS kernel [13, 41], and proving properties via formal verifications [52, 54]. Honeycomb utilizes techniques including security monitors and virtualization [10, 79] to remove the Linux kernel and the device driver out of the TCB.

*Software fault isolation (SFI).* Lightweight fault isolations [46, 57, 88] have been proven effective on the x86 architecture. Essentially, validation in Honeycomb is a form of SFI for GPU kernels. Honeycomb, however, combines the SFI with an alternative memory layout and other system-level supports to extend the fault isolation to a secure execution environment.

*Polyhedral analysis.* There is rich literature on utilizing polyhedral representations for loop analysis and transformations [8, 14, 35, 58]. Researchers have extended the approaches to more general cases [12]. Honeycomb uses the polyhedral analysis to model the effects of GPU memory access and to ensure that the memory access conforms with the security policy.

## 13 Conclusion

Honeycomb demonstrates that static analysis (validation) is a practical and flexible technique to enforce security for GPU applications. Combining with hardware and OS support, Honeycomb's validation guarantees powerful system-wide invariants like every memory access in the applications conforms

with the security policies. As a result, Honeycomb has reduced the size of TCB by  $18\times$ , and provided a secure IPC primitive that is  $529\times$  faster than conventional approaches.

The evaluation of Honeycomb on five representative benchmark suites, 23 applications in total, shows that Honeycomb is practical and efficient to provide secure GPU TEEs for real-world applications. It requires inserting few or none runtime checks into the GPU kernels to validate them, thus the runtime overhead is minimal. Large language model workloads like BERT and NanoGPT have  $\sim 2\%$  runtime overheads on Honeycomb.

The boom of GPU applications today requires continuous innovations in GPU software/hardware stack. Our experience on Honeycomb shows that static analysis has a lot of potential to help explore novel designs in the full software/hardware stack and to speed up innovations.

## Acknowledgments

We would like to thank our shepherd, Christopher Rossbach, and the anonymous reviewers for their comments and feedback on our work. We thank Quanxi Li, Shuoming Zhang for their contributions on an early implementation of this work. We also thank the Stanford Platform Lab and its affiliates. This work was partially supported by the National Key R&D Program of China (2021ZD0110101) and the National Natural Science Foundation of China (62072262, 62090024, 62232015).

## References

- [1] Advanced Micro Devices, Inc. AMD SEV-TIO: Trusted I/O for secure encrypted virtualization. <https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf>.
- [2] Advanced Micro Devices, Inc. HIP: C++ heterogeneous-compute interface for portability. <https://github.com/ROCm-Developer-Tools/HIP>.
- [3] Advanced Micro Devices, Inc. ROCm. <https://github.com/RadeonOpenCompute/ROCm>.
- [4] Advanced Micro Devices, Inc. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, 2020.
- [5] Advanced Micro Devices Inc. and Hewlett-Packard Inc. PCI express access control services (ACS), 2006.
- [6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 2007.

- [7] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *ASPLOS*, 2016.
- [8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *CGO*, 2019.
- [9] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *OSDI*, 2020.
- [10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.
- [11] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har’El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: Design and implementation of nested virtualization. In *OSDI*, 2010.
- [12] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, 2010.
- [13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN Operating System. In *SOSP*, 1995.
- [14] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, 2008.
- [15] Ben Boyter. Sloc Cloc and Code. <https://github.com/boyter/scc>.
- [16] Broadcom Inc. Videocore IV 3D architecture reference guide. <https://docs.broadcom.com/doc/12358545>.
- [17] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.
- [18] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *CCS*, 2008.
- [19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX Kingdom with transient Out-of-Order execution. In *USENIX Security*, 2018.
- [20] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.
- [21] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating speculative-execution attacks on SGX with hyperrace. In *DSC*, 2019.
- [22] Francis S Collins and Harold Varmus. A new initiative on precision medicine. *New England journal of medicine*, 2015.
- [23] CVE. CVE-2020-5991. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5991>.
- [24] CVE. CVE-2021-1098. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1098>.
- [25] CVE. CVE-2022-20186. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-20186>.
- [26] CVE. CVE-2022-21821. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21821>.
- [27] CVE. CVE-2022-31609. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31609>.
- [28] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. StrongBox: A GPU TEE on arm endpoints. In *CCS*, 2022.
- [29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [30] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *PLDI*, 2006.
- [31] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976.
- [32] Morris Dworkin, Elaine Barker, James Nechvatal, James Fote, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (AES), 2001.
- [33] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos Operating System. In *SOSP*, 2005.



- [34] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE S&P*, 2016.
- [35] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 2012.
- [36] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In *MICRO*, 2021.
- [37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
- [38] Hugging Face Inc. BERT base model. <https://huggingface.co/bert-base-uncased>.
- [39] Hugging Face Inc. GPT-2. <https://huggingface.co/gpt2>.
- [40] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE S&P*, 2013.
- [41] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 2007.
- [42] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *NSDI*, 2020.
- [43] Intel. Intel trust domain extensions. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>.
- [44] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *ASPLOS*, 2019.
- [45] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xipapu Luo, and Heming Cui. CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *MICRO*, 2022.
- [46] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Trust but verify: SFI safety for native-compiled Wasm. In *NDSS*, 2021.
- [47] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.
- [48] Andrej Karpathy. NanoGPT. <https://github.com/karpathy/nanoGPT>.
- [49] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasili Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. MIOpen: An open source library for deep learning primitives, 2019.
- [50] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *ISCA*, 2022.
- [51] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.
- [52] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.
- [53] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
- [54] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS*, 2013.
- [55] Microsoft Inc. Secure boot. <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot>.
- [56] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, 2020.
- [57] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.
- [58] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *ASPLOS*, 2015.

- [59] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, 2006.
- [60] George C. Necula. Proof-carrying code. In *POPL*, 1997.
- [61] Lucien K. L. Ng and Sherman S. M. Chow. GForce: GPU-friendly oblivious and rapid neural network inference. In *USENIX Security*, 2021.
- [62] NVIDIA Inc. FasterTransformer 5.3. <https://github.com/NVIDIA/FasterTransformer>.
- [63] NVIDIA Inc. Developing a Linux kernel module using RDMA for GPUDirect. Technical report, 2022.
- [64] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.
- [65] PCI-SIG. Integrity and data encryption (IDE) ECN. <https://members.pcisig.com/wg/PCI-SIG/document/16599>.
- [66] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving video analytics as a cloud service. In *USENIX Security*, 2020.
- [67] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J. Nair. Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking. In *ISCA*, 2022.
- [68] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [69] Adam Rodnitzky. Sensing breakdown: Waymo jaguar I-pace robotaxi. <https://www.tangramvision.com/blog/sensing-breakdown-waymo-jaguar-i-pace-robotaxi>, 2022.
- [70] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *SoCC*, 2021.
- [71] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MI-CRO*, 2021.
- [72] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA*, 2022.
- [73] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B. Schneider. Nexus: A new operating system for trustworthy computing. In *SOSP*, 2005.
- [74] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *ASPLOS*, 2013.
- [75] Paul Staat, Johannes Tobisch, Christian Zenger, and Christof Paar. Anti-tamper radio: System-level tamper detection for computing systems. In *IEEE S&P*, 2022.
- [76] Standard Performance Evaluation Corporation. The SPEC ACCEL benchmark suite. <https://www.spec.org/accel>.
- [77] Zhendong Su. Refutation unsoundness issue on a QF\_UFNIA instance. <https://github.com/Z3Prover/z3/issues/6693>, 2023.
- [78] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *IEEE S&P*, 2021.
- [79] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the Illinois Browser Operating System. In *OSDI*, 2010.
- [80] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 2019.
- [81] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.
- [82] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.
- [83] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI*, 2018.
- [84] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *USENIX Security*, 2022.
- [85] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021.
- [86] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

- [87] A. Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed DRAM rows. In *HPCA*, 2021.
- [88] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.
- [89] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *MICRO*, 2020.
- [90] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.
- [91] Kaihao Zhang, Dongxu Li, Wenhan Luo, Wenqi Ren, Björn Stenger, Wei Liu, Hongdong Li, and Ming-Hsuan Yang. Benchmarking ultra-high-definition image super-resolution. In *ICCV*, 2021.
- [92] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. AKG: Automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI*, 2021.
- [93] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. PPMLAC: High performance chipset architecture for secure multi-party computation. In *ISCA*, 2022.
- [94] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE S&P*, 2012.
- [95] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *IEEE S&P*, 2020.

# An Extensible Orchestration and Protection Framework for Confidential Cloud Computing

Adil Ahmad<sup>†</sup>, Alex Shultz<sup>†</sup>, Byoungyoung Lee<sup>\*</sup>, Pedro Fonseca<sup>§</sup>

<sup>†</sup>Arizona State University <sup>\*</sup>Seoul National University <sup>§</sup>Purdue University

## Abstract

Confidential computing solutions are crucial to address the cloud privacy concerns. Although SGX has witnessed significant adoption in the cloud, the reliance on hardware implementation is restrictive for cloud providers in terms of orchestrating deployments and providing stronger security to their clients' enclaves. eOPF addresses this limitation by providing a comprehensive, secure hypervisor-level instrumentation framework with the ability to monitor all enclave-OS interactions and implement protected services. eOPF overcomes several challenges including bridging the semantic gap between the hypervisor and SGX and attesting the co-location of the framework with enclaves. Using eOPF, we implement two protected services that provide platform resource orchestration and complementary enclave side-channel defense. Our evaluation shows that eOPF incurs very low performance overhead (<2%) in its default state and only modest overhead (geometric mean of 17% on SPEC) when strong, complementary side-channel defenses are enabled, making eOPF an efficient and practical solution for the cloud.

## 1 Introduction

The previous two decades have shown a substantial growth in internet services enabled by the cloud. Unfortunately, using cloud services requires users to outsource sensitive code, data, or both to cloud infrastructures shared by untrusted individuals. Moreover, the rise in cyber-attacks and corresponding increasing governmental regulations on sensitive information management (e.g., CCPA, GDPR) have made cloud privacy a first-order concern for many cloud providers and users. Thus, the cloud model success increasingly depends on providing strong privacy guarantees.

Cloud providers have been trying to accommodate the user demand for privacy using confidential computing solutions. Such solutions allow secure computation on cloud machines without trusting the machine's huge and vulnerable software codebase like the operating system (OS). Among several ap-

proaches, the hardware-protected Intel Software Guard eXtensions (SGX) *enclaves* have turned out to be the most popular key building block. In particular, SGX is already deployed by major cloud providers (e.g., Microsoft Azure [24], IBM Cloud [55]), thanks in no small part due to the extensive software ecosystem (e.g., development kits and library OSs) that aids the development of new SGX programs and porting existing codebases [17, 25, 76, 83].

Despite the strong security properties of SGX, its inflexible hardware implementation poses pragmatic challenges for cloud providers and users. For instance, modern cloud services aim to be elastic, which often comes with a pay-as-you-go model that requires detailed fine-grained resource usage accounting. Unfortunately, SGX only provides detailed enclave-usage data to the OS, which is untrusted even when cloud providers run containerized instances since the OS' large codebase is susceptible to attacks from untrusted users on the machine. Moreover, since SGX's inception, many attacks have been discovered against enclaves, which are currently difficult for cloud providers to mitigate. In particular, hardware updates for several attacks (e.g., digital side-channels) were never implemented by Intel, eroding user trust in the security capabilities of SGX and exposing users to attacks.

This paper proposes eOPF, a framework designed to provide a privileged trusted software environment for cloud providers to deploy secure services on enclave-running platforms. eOPF leverages virtualization extensions to enable trustworthy and complete interposition between enclaves and the OS. By virtue of such interposition, eOPF allows cloud providers to build protected services that enhance enclaves. In particular, this paper shows how eOPF can be used to (a) securely orchestrate enclaves (e.g., control and monitor enclave resource usage) and (b) add complementary enclave side-channel defenses.

Leveraging a framework like eOPF to enable services for enclaves poses several technical challenges. First, to enable protection and resource monitoring, the framework should interpose between the OS and enclave and mediate all OS-enclave interactions. Unfortunately, this capability is not na-



tively available to the virtualization layer. Second, for remote users to trust that their enclaves are protected, they should determine that their enclaves are co-located with the framework. Unfortunately, there is currently no mechanism to guarantee such co-location. Third, even if previous challenges are solved, it is necessary to show how to securely implement orchestration and protection services using the framework.

eOPF achieves complete interposition of all enclave-OS interactions through a combination of hardware-enabled interception features and several indirect mechanisms (§4.1). In particular, Intel CPUs allow a virtualization-based framework to intercept all SGX supervisor instructions, which are used to manage enclave creation and destruction. To reliably trap on all events during enclave execution (e.g., enclave start and stop events), eOPF carefully leverages a combination of memory protection (namely extended page tables), the x86 single-step mode, and interrupt-interception mechanisms.

We address the co-location challenge by designing, to our knowledge, the first platform-enclave co-attestation protocol allowing enclave users to trust that their enclaves are protected (§4.2). Instead of naively leveraging the virtualization framework for enclave installation, which does not prove co-location to a remote user, eOPF leverages a combination of the cloud provider’s initial provisioning, intercepted enclave installation, and SGX remote attestation to achieve co-location guarantees for cloud users.

In its current form, eOPF includes a library of functions to allow cloud providers and users to enable several orchestration and protection services (§5). For instance, eOPF implements a library of side-channel defenses that users can select during runtime. The defense capabilities are implemented at a resource-level (e.g., page tables, caches) in a principled manner to isolate resources responsible for side-channel and ensure full protection. Additional services can be flexibly implemented through further software libraries.

We implemented a proof-of-concept eOPF framework with services on the Bareflank extensible framework [3]. In addition, we analyze the end-to-end security of the system and show that eOPF is effective at preventing diverse attacks against its interposition, co-attestation, and implemented services. Furthermore, we demonstrate eOPF’s performance (§8) using benchmarks and real-world programs—the SPEC CPU 2006 integer suite [13], Redis [12], and Lighttpd [9]. Our results indicate that the base framework (without side-channel defenses) incurs less than 2% performance impact to enclaves, and when all side-channel defenses are enabled, it incurs a geometric mean performance overhead of 17%, hence suitable for diverse use-cases in today’s clouds.

## 2 Confidential Cloud Computing

This section describes the confidential cloud computing system model, threat model, and research goal of eOPF. Fig. 1 provides an overview of the system model.

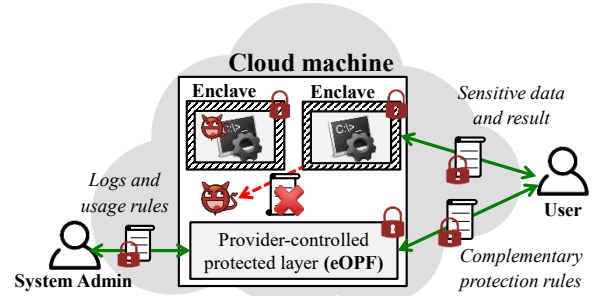


Figure 1: Our confidential computing model.

### 2.1 System Model

We assume that users want to run sensitive computations on the cloud (e.g., healthcare analytics on genetic information of several individuals [16]). They trust the cloud provider (like other confidential computing approaches [2, 47, 48]) but do not trust other users on the machine. The cloud provider does not trust users and aims to protect users from each other, since some may be malicious.

The cloud provider leverages SGX to enable users to securely run computations in enclaves, without trusting the bulk of the software stack or other users. SGX has several advantages over other approaches. First, SGX provides strong confidentiality and integrity guarantees against a wide-range of attacks [33]. Second, SGX is now widely-available in Intel server machines [4], a sizeable portion of all servers in the market today. Third, there are mature software development kits (SDKs) allowing users to port their programs to SGX enclaves [17] and library operating systems [76, 83] that allow users to easily run legacy programs inside enclaves.

Our model also assumes that the cloud provider runs a type-1 hypervisor on the machine (e.g., AWS Nitro [2], protected KVM [54]) and provisions containerized instances for users. Type-1 hypervisors provide better security guarantees due to a thin software codebase running at the virtualization layer (i.e., Intel VMX [52]). Containerized instances increase resource efficiency and simplify resource provisioning; hence, containerized instances underlay increasingly popular cloud models, such as microservices and serverless computing [1, 11, 72]. Moreover, since users run their sensitive computations inside SGX enclaves, the traditional isolation limitations of container instances do not apply. Nevertheless, our model also directly applies to scenarios where the cloud provider provisions virtual machines (VMs) (§9).

Since the cloud machine runs enclaves of different users, the cloud provider needs to deploy a *flexible, protected* layer to easily manage enclave instances, including managing resource oversubscription (e.g., AWS burstable instances [6]) to maximize resource efficiency. Furthermore, the cloud provider wants to use this layer to offer *enhanced, complementary protection* for enclaves against attacks that SGX does not protect, potentially by charging a higher cost.

## 2.2 Threat Model and Assumptions

The cloud provider and honest users assume that a dishonest user (or other third-parties) may compromise the machine's operating system, by leveraging a kernel vulnerability or mis-configuration. After OS compromise, they assume that an attacker will launch attacks to (a) steal sensitive information from enclaves using digital side-channels [40, 43, 60, 84, 88] or (b) launch attacks against the platform or other users using enclaves (e.g., to prevent malware introspection [74]).

**Assumptions.** This heading describes our assumptions about the security of the SGX processor and hypervisor, as well as the availability of a trusted key management service.

*SGX processor.* We trust that the processor is correctly implemented. In particular, it correctly prevents direct access of enclaves from external software and implements all cryptographic and remote attestation primitives.

*Hypervisor.* We trust the hypervisor is correct and securely initialized on the cloud machine by the trusted cloud provider. A cloud provider can securely initialize a hypervisor by leveraging UEFI secure boot [87] or verified late launch (e.g., Intel TXT [52, 63]). Leveraging a trusted platform module (TPM) [23], the provider can also attest the correct initialization remotely. Note that although we trust the hypervisor, its compromise cannot harm existing SGX guarantees since enclaves are protected from hypervisors. Please refer to §7.3 for a hypervisor TCB discussion.

*Key management service.* We assume the availability of a trusted local or remote key management service (KMS). A trusted local key management service can be designed using a TPM. In either scenario, we assume that our system has secure access to the KMS (e.g., using an isolated channel to a local device [89] or authenticated encrypted channel).

**Out-of-scope.** We do not consider attacks through micro-architectural defects, software vulnerabilities inside enclave programs, system calls, and physical attacks. We also exclude attacks through micro-architectural defects (e.g., speculative execution attacks [29, 56, 85]). Defenses enabled by our system (§5.2) for side-channels also prevent the exploitation of micro-architectural defects [27, 56] in SGX enclaves through these channels. However, the root cause of micro-architectural defects are hardware bugs, and as such they are already routinely addressed by Intel through microcode or hardware updates [7, 51]. Existing schemes [57, 75, 76] can prevent vulnerability exploitation in buggy enclave programs and protect enclaves from malicious system call results [25, 49]. Finally, physical attacks that infer DRAM access patterns and electromagnetic analysis are very expensive [58].

## 2.3 Research Goal

Given the mistrust of the OS, this paper's research goal is to design a hypervisor-level instrumentation framework that

allows cloud providers to enable protected services on enclave platforms. The framework is designed to be flexible and support two use-case classes: (a) secure enclave orchestration (e.g., preventing dishonest users from running enclaves, monitoring enclave resource usage) and (b) complementary side-channel defense for enclaves (e.g., by isolating resources).

Combining a hypervisor-level framework with SGX is favorable for cloud providers and users. From a cloud provider's perspective, hypervisor-only approaches [47, 48] offer more control but they require significant investment to design in-house full enclave abstractions and implement the corresponding SDKs. From a user's perspective, hypervisor-only approaches offer flexible functionality (e.g., resource isolation) but they have a single point-of-failure (i.e., the cloud hypervisor) in terms of data protection. Our approach solves both problems by leveraging SGX with its robust software ecosystem [17, 76, 83] and complementary data protection guarantees in the event of a cloud hypervisor compromise. Hence, co-leveraging SGX and a hypervisor is a *best-of-both-worlds* scenario for cloud providers and users.

## 3 Background on Intel SGX

Intel SGX [64] allows a process to create protected execution contexts called *enclaves*. This section describes memory protection, lifecycle, and remote attestation aspects of SGX since they are relevant to eOPF.

**Enclave page cache (EPC).** This is a reserved physical memory region where enclaves reside. SGX relies on the operating system to over-subscribe the EPC using *demand paging* (i.e., securely retrieving pages from an encrypted backing store using page faults and updating page tables).

**Enclave lifecycle.** An enclave is created by the OS using SGX supervisor leaf instructions (ENCLS). During enclave execution, the untrusted and enclave parts of the process execute SGX user leaf instructions (ENCLU) for a world switch.

*Enclave Creation.* The OS executes ECREATE to create an enclave context. After context creation, the OS invokes EADD to copy initial code and data, provided by the user, from non-enclave to enclave pages. Then, the OS executes EEXTEND to measure the copied page (explained in the next section). Finally, the OS executes EINIT to finalize the enclave.

*Enclave Entry/Exit/Resumption.* The untrusted part of the process can transition to the enclave mode using EENTER. Afterward, the enclave executes EEXIT to transition back to the untrusted mode, for two reasons: (a) synchronous exits (i.e., to perform a system-call or shutdown the enclave) and (b) asynchronous exits (i.e., to handle page faults, interrupts, and exceptions). After handling the reason for an exit, the process executes ERESUME to resume the enclave.

**Remote attestation.** SGX enables remote users to assert that their code and initial data is correctly loaded into an enclave

by sending them the enclave measurement (MRENCLAVE or  $M_e$ ) signed using the SGX CPU’s attestation key.

The enclave measurement process is entirely deterministic [30]. The measurement algorithm has an initialization, update, and finalization stage. During initialization (ECREATE), the CPU creates an initial SHA-256 hash using the OS-provided SGX Enclave Control Structure (SECS), which contains the enclave’s metadata (e.g., base address and size). In the update stages, the CPU updates the hash using each page added into the enclave (at EADD) alongside an OS-provided security information (SECINFO) block. The SECINFO block contains information about the page’s metadata (e.g., offset and permissions). In the same stage, the CPU measures each added page in 512-bit blocks (at EEXTEND). Lastly, in the finalization stage, the enclave’s measurement is hashed one last time with the total count of bits that are updated in the MRENCLAVE (at EINIT).

In formal terms, assuming an enclave of  $N$  pages ( $P^1$  to  $P^N$ ) with  $Z$  total bits, the entire enclave measurement is:

$$M_e = H_{fin}(H_{upd}(\dots H_{upd}(H_{upd}(IV, SECS), P^1) \dots, P^N), Z)$$

In this equation,  $IV$  are the initialization vectors. Additionally, for simplicity, we assume that  $H_{upd}(state, P)$  also includes a hash of the SECINFO of page  $P$ .

## 4 eOPF Design

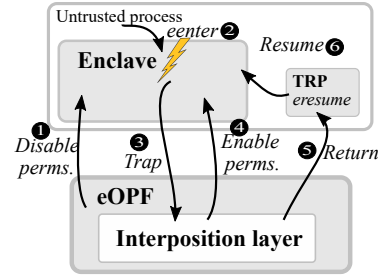
eOPF provides a privileged trusted software environment for cloud providers to build protected services on their SGX-compatible confidential computing platforms. eOPF leverages hypervisor-level instrumentation to enable trustworthy and complete interposition between enclaves and the OS. This interposition allows users to run protected services that augment enclave security and improve resource management (e.g., measure enclave execution time).

Hypervisor-level or virtual machine extensions (VMX) [52] allow eOPF to monitor and control the execution of the OS, e.g., observe and manipulate page tables. In particular, by leveraging VMX, eOPF can intercept supervisor instructions executed by the OS and exceptions raised by the machine. Moreover, eOPF can also leverage VMX features to protect its TCB from the OS and external devices.

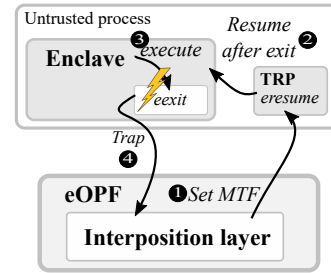
Designing a secure hypervisor-level instrumentation framework for enclaves poses several challenges that we address:

**C1: Semantic VMX-SGX gap.** Complete and reliable interposition of enclave interactions is needed to build protected services. While VMX framework can natively trap on SGX supervisor instructions for enclave management, it cannot natively trap SGX user instructions that determine when an enclave starts or stops. These latter events are typically junctions of information exchange between enclaves and the OS; hence, interposition is critical to augment enclave security.

**C2: Co-location attestation hurdle.** SGX’s remote attestation allows a remote user to know that their programs are



(a) EPT-based enclave entry and resume monitor



(b) Single-step-based synchronous enclave exit monitor

Figure 2: eOPF’s interposition on enclave entries and synchronous enclave exits.

running inside an enclave, but provides no guarantees that this enclave is running on the cloud provider’s machine. Without such guarantees, users cannot tell that their enclaves are protected by eOPF.

**C3: Practical service libraries.** It is necessary to show how to leverage the enclave instrumentation framework to build protected services. To help users easily build such services, it is necessary to design and implement easy-to-use libraries with core functions (e.g., transparently augment enclave protections against classes of attacks).

### 4.1 Enclave Life-Cycle Interposition

eOPF achieves complete interposition over all interactions between an enclave and the OS using native x86 features and new indirect interposition mechanisms.

**Enclave management monitor.** eOPF leverages the native capabilities of x86 virtualization to trap all SGX supervisor instructions (ENCLS), which are used for enclave creation, deletion, and other management tasks. In particular, eOPF sets the ENCLS-interception bit and its corresponding instruction bitmap in the x86 Virtual Machine Control Structure (VMCS) [52] to trap ENCLS instructions. On a trap, eOPF undertakes three sequential steps. First, eOPF implements service-specific operations needed for the instruction (refer to §4.2 and §5.1). Second, eOPF executes the trapped instruction using its trusted code and parameters provided by the



OS. Third, eOPF resumes the OS' execution from after the instruction by updating the processor's program counter.

**EPT-enforced enclave entry and resume monitor.** eOPF tracks enclave entry and resume events using the extended page tables (EPT). EPT allows a virtualization framework to protect regions of the physical memory from unauthorized read, write, and execute operations. By removing execute permissions from the enclave page cache (EPC) region, eOPF can ensure that every time enclave code is executed it raises a trap. The challenge, however, is that the trap is raised as an enclave exit, and there is no guarantee that the OS will resume the enclave after eOPF resolves the trap.

eOPF addresses the challenge by creating a trusted resume pointer (TRP), a reserved location within the process' address space that is guaranteed to execute `ERESUME`. eOPF inserts the TRP at a location where it does not significantly impact the OS' process memory management (i.e., only shares top-level page table with the remaining addresses). The OS is notified through a shared memory channel and kernel module (§6) to reserve the TRP region. eOPF write-protects the TRP and page tables that address this location using EPT, ensuring the TRP cannot be modified by the OS.

Fig. 2-(a) illustrates the EPT-based enclave entry and resume scheme employed by eOPF. On every processor core, eOPF leverages the EPT to disable execute permissions for all enclave page cache (EPC) regions (1). Hence, when a process transitions into the enclave region (i.e., using `EENTER` or `ERESUME`) (2), the CPU traps the operation with an EPT violation (3). eOPF resolves the violation by enabling execution permissions (4) and redirecting the program counter (`rip`) to the TRP (5). Finally, the enclave resumes (6).

**Dual enclave exit monitors.** eOPF uses the x86 single step mode and interrupt interception features to track synchronous and asynchronous enclave exits, respectively.

*Single-step-based synchronous exit monitor.* eOPF leverages the x86 single step mode to trap synchronous exits (e.g., for an exit-based system call). In particular, since system software is not allowed to intercept enclave execution apart from debug mode, the execution (from `EENTER` to `EEXIT`) within an enclave is considered a single step [32].

Fig. 2-(b) illustrates the synchronous exit monitor process during an exit-based enclave system call. eOPF enables the single-step mode by setting the MTF in the current processor's VMCS (1) before entering the enclave (2). Hence, the processor's execution traps to eOPF's monitor when the enclave executes `EEXIT` (3~4). eOPF disables this trap allowing the exit to be processed by the system. This process is repeated at the next enclave entry.

*Interrupt-based asynchronous exit monitor.* Apart from synchronous exits, the enclave performs asynchronous exits in order to service interrupts (e.g., raised by the timer hardware). eOPF ensures that all interrupts are trapped by setting the interrupt-interception bit inside the VMCS.

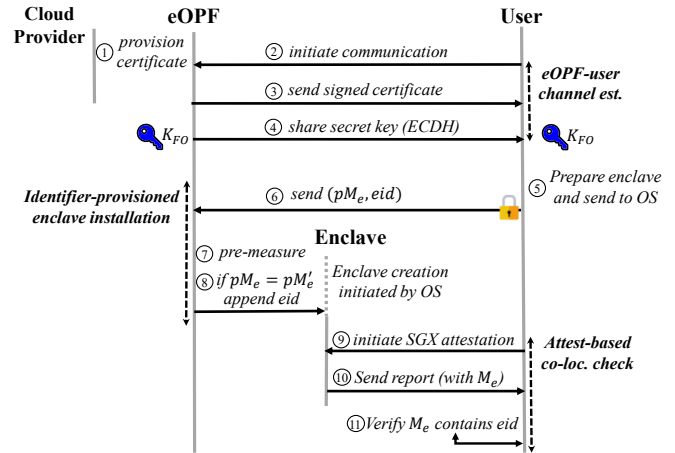


Figure 3: The platform-enclave co-attestation protocol.

## 4.2 Platform-Enclave Co-Attestation

SGX's attestation does not tell a user that their enclaves are running on their cloud provider's machine. In particular, the attestation report only contains information about the platform's security version (microcode) [53]. This is a significant challenge that motivates eOPF's co-attestation protocol.

Without a guarantee of co-location between enclaves and eOPF, an attacker (e.g., a malicious user) who has compromised a cloud machine's operating system could trick users into sending data to enclaves unprotected by eOPF. In particular, the attacker could exfiltrate a user's code from a cloud machine, send it to their own SGX-capable machine, and install it inside an unprotected enclave. Afterwards, the attacker could route all network traffic from the cloud machine to their own machine, and trick the user into sending their confidential data to the unprotected enclave. This potential attack would not require collusion with the VMM, since VMMs must allow network traffic to a cloud user's machine.

A naive approach to prevent this attack would be to leverage eOPF's interposition (last section) and design an entirely new in-house SGX attestation approach. Unfortunately, that is a significant undertaking that would require complex attestation functionality to be redundantly re-implemented and make enclave security completely reliant on eOPF, instead of complementary to SGX protections.

eOPF implements a more secure, novel co-attestation protocol that leverages both eOPF's interposition and SGX remote attestation. The key insight of our approach is that eOPF's interposition allows it to bind an *infeasible-to-guess* secret (as a watermark) to a user's enclave created on its machine, which will be transmitted and validated through SGX remote attestation to the remote user.

The eOPF co-attestation protocol has three stages. First, with the help of the trusted cloud provider, a remote user establishes a secure communication channel with an eOPF instance. Through this channel, the user sends a secret to this eOPF instance. Second, during enclave creation, the eOPF



instance securely and transparently inserts that secret into initial enclave memory. Third, the remote user leverages SGX attestation to verify the initial enclave contents and validate that co-attestation secret is valid, thereby confirming that the enclave is co-located with a cloud provider eOPF instance. In all these steps, eOPF uses a side-channel resistant cryptographic library (e.g., EverCrypt [69]) to protect secret keys. Fig. 3 illustrates our co-attestation protocol.

**eOPF-user channel establishment.** With the provider’s help, an eOPF instance on a cloud machine and a user of the machine establish a secure communication channel. In particular, during initial platform provisioning, the cloud provider installs the eOPF framework on the machine with a signed digital attestation certificate (①). This certificate and the corresponding private key is securely stored by eOPF using a trusted key management service (e.g., protected storage device or a trusted platform module) (§2.2).

When a remote user wants to run enclaves on the machine, the user will first establish a secure communication channel with the eOPF framework (②–④). In particular, the user asks the framework to authenticate itself (②) and the framework responds with its signed certificate (③). If the certificate signature is valid—based on the cloud provider’s off-the-band provided public key ( $PubK_C$ )—the remote user and the eOPF instance establish a shared secret key ( $K_{FO}$ ) (④). Note that eOPF does not require direct access to the network. In particular, all eOPF-user communication can be routed through the operating system. This approach is safe since all communication after shared channel establishment is end-to-end encrypted (using  $K_{FO}$ ) and is similar to how enclaves use the operating system as an untrusted network transport.

**Identifier-provisioned enclave installation.** Once a secure channel between the user and an eOPF instance is established, the eOPF instance installs a secret identifier into a user-specified enclave during enclave creation. We explain this process in the next paragraphs.

The user compiles a special enclave binary with one empty reserved memory page (4KB) at the end using a custom linker script and sends it to the OS. The reserved memory page will be used to hold a random 4KB secret (called  $eid$ ). The user also creates a *premeasurement* ( $pM_e$ ) of this enclave binary. The  $pM_e$  is a hash of all enclave binary pages using SGX’s enclave measurement algorithm (described in §3) except the last reserved page. In formal terms, assuming the enclave has  $N$  pages ( $P^1$  to  $P^N$ ), the  $pM_e$  is calculated as follows:

$$pM_e = H_{upd}(\dots H_{upd}(H_{upd}(IV, SECS), P^1), \dots, P^{N-1})$$

The user sends the enclave binary to the OS (⑤). Simultaneously, the user sends the  $pM_e$  and  $eid$  to eOPF using their secure communication channel (⑥).

During enclave creation, eOPF recreates  $pM_e$  to attest that the correct user enclave is being initialized on the machine (⑦). In particular, on enclave creation (§4.1), eOPF recreates the hash using an internal SHA-256 library config-

ured with the OS-provided parameters to `ECREATE` and `EADD` instructions (i.e., `SECS`, `SECINFO`, and page contents). If the premeasurement matches  $pM_e$ , eOPF transparently modifies the last enclave page to include the  $eid$  (⑧). This requires trapping `EADD` and replacing the contents inside the physical page being added to the enclave.

There are three requirements for the above operations to securely happen. First, the OS should not modify an enclave page while it is being measured by eOPF. Second, the OS should not read the  $eid$  while it is being copied into the enclave. Third, after copying  $eid$ , there should be no additional pages added to the enclave. eOPF fulfills the first two requirements using EPT. In particular, eOPF write-protects the `SECINFO` and page contents of the enclave page before the premeasurement process. Similarly, while adding  $eid$  to the reserved page, eOPF removes all permissions from the page before executing `EADD`. These protections are only disabled after `EADD` executes (§4.1). Finally, eOPF does not allow any `EADD` operation on the enclave after adding  $eid$ , ensuring that it really is the user’s enclave, and not a malicious enclave designed by the OS to steal  $eid$ .

Please refer to §9 for a discussion on how this co-attestation step can be potentially achieved without premeasurement.

**Attestation-based co-location check.** Once the enclave is securely provisioned with a secret identifier ( $eid$ ) that is only known to the eOPF instance, a remote user can leverage SGX remote attestation (§3) to check whether their enclave contains that identifier or not (⑨~⑪). In formal terms, assuming a correct enclave page with  $eid$  is  $P^{eid}$  and  $Z$  total bits, the correct enclave measurement should be as follows:

$$M_e = H_{fin}(H_{upd}(pM_e, P^{eid}), Z)$$

Since  $eid$  is a 4KB identifier, there is an infinitesimally small chance for an attacker to randomly guess ( $2^{-32768}$ ); hence, this measurement can only hold if the enclave memory contains  $eid$  provisioned by eOPF, proving co-location.

## 5 eOPF Protected Services

eOPF allows cloud providers to implement protected services in an extensible manner. This section demonstrates eOPF’s value by presenting the design of two services that help cloud providers manage resources and augment enclave security.

### 5.1 Secure Enclave Orchestration

The secure enclave orchestration service gives cloud providers the ability to control what enclaves run on their platform, detect when enclaves are used for malicious purposes, and obtain detailed enclave-related resource usage for accountability and billing. This section describes how eOPF allows cloud providers to achieve such orchestration.

**Protected launch control.** eOPF ensures that only users approved by the cloud provider are allowed to run enclaves on

cloud machines. In modern SGX machines, cloud providers leverage flexible launch control (FLC) [53] to provision a platform and provide launch tokens to their customers without relying on Intel’s provisioning service. Unfortunately, flexible launch is controlled by the untrusted OS using MSR, IA32\_SGXLEPUBKEYHASH{0–3} and the attacker can exploit this feature to launch arbitrary enclaves. eOPF leverages virtualization features to trap all writes to MSRs (i.e., WRMSR) and disallows modifications to launch control MSRs. Hence, all valid changes to the FLC feature must come from the cloud provider directly to eOPF.

**Malware scanning.** An attacker may try to hide malware on the cloud machine using shielded environments like enclaves [39, 74]. For instance, research shows that attackers can use TPMs to hide attack targets from forensic analysts [39]. A typical approach to detect malware on a machine is by scanning binaries and signature matching against a database of known malware. Although a simple approach, this is effective in practice (e.g., one study shows 59% of known malware can be detected by signature matching tools [81]).

eOPF enables secure scanning of enclave contents within its framework during enclave creation. In particular, during enclave creation, as each page is being added to the enclave (§4.1), eOPF compares the hash of contents against known malware hashes. eOPF also provides the ability to prevent the attacker from installing a barebones enclave and leveraging it to insert malware (e.g., by enabling execute permissions on data pages). This is achieved by intercepting and rejecting `EMODPE`, an `ENCLS` leaf instruction leveraged for changing existing enclave page permission changes and adding additional pages.

One concern with enclave content scanning is user privacy especially in scenarios where enclave code is an intellectual property (e.g., services like 23andMe [16] with proprietary healthcare analysis algorithms). Such concerns can be mitigated if the cloud provider runs their scanning tool inside an enclave and makes the source code of the scanner publicly-available for enclave attestation by remote users. If a proprietary scanner is used, the provider can employ SGX sandbox enforcement mechanisms [19, 49]. With these, users can trust that the proprietary scanning tool will be unable to leak sensitive information from the scanning enclave.

**Resource usage statistics.** Once an allowed enclave is running on the machine, eOPF collects detailed statistics about the enclave’s machine resource usage and periodically sends it to a system administrator.

By default, eOPF collects information about two resources: CPU time and memory. In particular, eOPF collects how much time (in cycles using `RDTSCP`) is dedicated to the user’s enclaves by implementing timers at enclave entries and exits. To prevent the OS from modifying CPU timer information, eOPF disallows all changes to timer-related MSRs [52]. eOPF also collects how much memory is allocated to the en-

clave. This is achieved by monitoring enclave page addition (`EADD`) and enclave page removal (`EWB`) instructions.

If a user enables complementary enclave side-channel defense, enclaves use additional resources (§5.2). eOPF also collects statistics of such usage for reporting purposes. In particular, eOPF tracks whether hyperthreading is disabled on a CPU core to defeat per-core side-channels. If the user selects static memory allocation for paging side-channel defense, this information is also collected. Finally, eOPF reports whether the enclave is using an isolated last-level cache or not, and how many partitions within the LLC are reserved for the user.

## 5.2 Complementary Side-Channel Defense

This service allows users to enable complementary principled defenses against digital side-channels. Digital side-channel attacks allow untrusted software on a machine (e.g., the OS) to observe the interactions of trusted software and the hardware platform [70]. Observation allow attackers to infer memory access patterns of an enclave program, which has been shown to leak sensitive enclave data (e.g., cryptographic keys) because many programs have data-dependent pathways [26].

To reason about defeating side-channels, we divide hardware resources based on how they can be observed (hence, exploited) into cross-core and per-core resources. For instance, last-level cache is shared by all processor cores, hence it can be observed by attacker on any core, while the L1/L2 caches are private to each processor core and can only be observed if the attacker runs code within the same core.

Using our classification and by integrating techniques from literature [60, 61, 66, 67], this service offers principled side-channel defense that can be flexibly enabled by users with minimal effort. In particular, the service isolates cross-core resources, ensuring an attacker cannot simultaneously observe enclave access onto the resource from any other core. Moreover, the service invalidates or deactivates per-core resources to ensure an attacker is unable to observe enclave access semantic when they run sequentially on a processor core after an enclave, or parrallely on an enclave-running core.

### 5.2.1 Cross-Core Resource Isolation

**Page tables.** The page table is created and maintained by the untrusted OS. The OS can infer page-granular (4KB for SGX enclaves) access patterns of an enclave through an enclave’s page tables. In particular, the OS can modify the enclave’s page tables to induce page faults [88] or stealthily observe the access bits of the enclave’s page table entries [84]. To avoid these attacks, eOPF allows the OS to create and delete the page tables, at enclave creation and deletion, respectively. However, eOPF prevents modifications to the page tables during enclave execution. Therefore, eOPF employs *temporal isolation* to protect the page tables.

After enclave creation, eOPF write-protects an enclave’s page tables using EPT. During each enclave entry, eOPF also checks the CR3 value to ensure that the OS did not try to create duplicated enclave page tables. Hence, eOPF ensures that the attacker cannot induce enclave page faults during execution. Furthermore, eOPF scans the enclave’s page tables and sets the access bit of each entry, ensuring that the attacker cannot leak information through access bits. At enclave shutdown, eOPF disables write-protection to let the OS handle page table deallocation. Please refer to §9 as to how this defense can be extended to support oblivious page swapping.

**Last-level cache (LLC).** The LLC contains cache lines from all programs executing on all processor cores. Hence, the LLC is vulnerable to cache attacks [26, 74]. To defeat these attacks, eOPF partitions the LLC such that an enclave’s cache lines are *spatially isolated* from untrusted programs.

Cache Allocation Technology (CAT) allows isolating cache lines of different CPU processors across different partitions in the LLC. Leveraging CAT, eOPF divides the LLC into enclave and non-enclave partitions. At enclave entries and resumes, eOPF switches the processor to the enclave partition, while the untrusted software (on other processors) use the non-enclave partition. On enclave exits, eOPF reverts the processor back to the non-enclave partition. While each partition can support unlimited enclaves, CAT can only support 15 distrusting partitions concurrently at this time. In particular, the latest CAT implementation has 16 domains [52] and 1 partition must remain reserved for untrusted software. In the future, if additional domains are implemented, eOPF can support additional distrusting partitions.

eOPF creates new LLC partitions using CAT-related MSRs, IA32\_L3\_MASK\_N. A processor follows the partition specified in its register IA32\_PQR\_ASSOC. Whenever a partition is changed, all cache-lines must be invalidated to enforce the change. eOPF achieves this using WBINVD. Furthermore, eOPF prevents modifications to CAT MSRs during enclave execution to ensure full control over CAT.

## 5.2.2 Per-Core Resource Invalidation and Deactivation

**Intra-core computational units (ICUs).** Such units include Arithmetic Logic Units (ALUs) and Translation-Lookaside Buffer (TLBs). An attacker can abuse hyper-threading, a hardware feature that allows concurrent execution of two threads on the same processor core, to infer an enclave’s access semantics onto ICUs [21, 44]. To defeat these attacks, eOPF ensures that hyper-threading is *deactivated* on the processor core that is running an enclave.

eOPF notifies the OS using its shared memory channel (§6) that a certain enclave should execute without hyper-threading. The OS can disable hyper-threading in software (e.g., OpenBSD does this by default [10]) by programming the x86 Local APIC [52]. In particular, once hyper-threading is disabled on a processor, it does not raise hardware interrupts.

Hence, eOPF monitors each core for hardware interrupts and if it observes hardware interrupts on enclave-running (hyper-threaded) processor cores, it terminates the enclave. On each enclave exit, ICUs are automatically flushed by the SGX processor, leaving no observable intermediate effect.

**L1 and L2 cache.** Enclave and untrusted programs that run sequentially or in parallel (using hyper-threading) on a processor core share cache lines across the L1 and L2 caches. An attacker can exploit this sharing to leak enclave contents through cache attacks [26, 43]. eOPF *deactivates* hyper-threading (previous section) to prevent parallel attacks. To protect against sequential attacks, eOPF *invalidates* the L1/L2 cache (using WBINVD) at enclave exits. Hence, all enclave contents are flushed back to memory and the attacker observes an empty cache state on each attack.

**Branch predictor units (BPUs).** Branch predictor units like the branch target buffer (BTB) and pattern history table (PHT) predict the control-flow of a computation in an out-of-order CPU. Attackers can use them to infer an enclave’s control-flow by observing whether a particular branch was taken or not [40, 60]. Unfortunately, the SGX CPU does not provide native mechanisms to invalidate these units. Nevertheless, eOPF deactivates the components critical for their side-channel exploits and designs software invalidation.

Prior work has shown that reliable attacks on the BTB require specialized units such as the Last Branch Record (LBR) or Intel Processor Trace (PT) [60], particularly because of the BTB’s small size in comparison to other predictors. The LBR and PT are performance tools that cannot be used by enclaves. Hence, eOPF deactivates the LBR and PT by setting MSRs, IA32\_DEBUGCTLA and IA32\_RTIT\_CTL, respectively, and denying all modifications to these MSRs.

eOPF uses knowledge of the PHT’s structure to implement a software invalidation technique, ensuring the attacker is unable to observe intermediate enclave artifacts on the PHT. In particular, the PHT contains 16,384 entries and is indexed by the lowest  $\log_2 N$  bits of a conditional branch instruction’s address [40]. Each PHT entry is a 2-bit Finite State Machine with 4 states: (a) Strongly Not-Taken, (b) Weakly Not-Taken, (c) Weakly-Taken, and (d) Strongly-Taken. An entry is updated each time the processor takes (or does not take) a branch. Using this knowledge, eOPF generates conditional branches aligned to each PHT entry. For each branch, the code performs an always-true arithmetic comparison and takes the branch. Therefore, each PHT entry moves towards the Strongly-Taken state. eOPF runs the code thrice to ensure that the final state of each PHT entry is Strongly-Taken. Hence, the attacker always observes a uniform state of the PHT.

## 6 Implementation

We built a prototype of eOPF using the Bareflank extensible framework [3]. However, in practice, eOPF can be built using



any VMX framework or type-1 hypervisor (§2.1). By default, eOPF enables EPT protections, traps all enclave events and critical processor-related events (e.g., WRMSR), and enables the enclave orchestration service (refer to §4.1 and §5.1). Apart from bootstrapping and VMX-specific code, the base framework includes a SHA-256 hash generator [65] for co-attestation and kernel module for eOPF-OS communication. Furthermore, we implemented three eOPF modules: a *paging module* (PM) for page table protections, a *caching module* (CM) for L1/L2 and last-level cache protections, and a *branching module* (BM) for BPU protections.

Our prototype does not currently implement communication with the user. Such communication is a one-time cost at enclave creation; hence, it does not impact runtime results. Also, Bareflank does not currently support IOMMU to protect the framework against device-based attacks. Nevertheless, IOMMU should be enabled by default in cloud machines and it is not an additional slowdown factor incurred by eOPF.

## 7 Security Analysis

This section provides a security analysis of eOPF and protected services by discussing several attacks and implemented defenses (Fig. 4). It concludes with a brief TCB discussion.

### 7.1 Analyzing Framework Security

**Preventing attacks against interposition.** eOPF requires secure interposition of enclave and important system events. To prevent this interposition, the attacker can try to overwrite the VMCS, a data structure that contains the ENCLS-interception bitmap, the monitor trap flag (MTF), and is responsible for enabling other important interception functionality (e.g., WRMSR traps). Moreover, the attacker can try to modify the TRP and trick eOPF into thinking the enclave resumed. eOPF prevents all aforementioned attacks (§4.1).

The VMCS and its extended instruction bitmaps are located in protected eOPF memory and the attacker is unable to modify these structures to disable ENCLS or other system functionality interposition. The protected memory is created from virtualization extensions. In particular, eOPF uses EPT protections to prevent software access and IOMMU protections to prevent device access [89]. The critical data structures (or tables) of EPT and IOMMU are also stored within the protected memory; hence, the OS cannot access them. Finally, the TRP is located in a reserved region of the enclave process' address space, it cannot be overwritten due to memory protections, and its page tables are also write-protected.

**Preventing attacks against co-attestation.** The attacker can attempt to circumvent co-attestation by trying to leak secret eOPF keys provisioned in the machine or the enclave unique ID (*eid*), trying to guess *eid*, and replaying communication. eOPF prevents all such attacks (§4.2).

| Potential attacks                           | eOPF defense                                                            |
|---------------------------------------------|-------------------------------------------------------------------------|
| <b>Against interposition (§4.1)</b>         |                                                                         |
| Modify VMCS                                 | Prevent software and device access using EPT and IOMMU, resp.           |
| Disable mem. protections                    | Access-protect EPT/IOMMU structures                                     |
| Modify TRP                                  | Write-protect TRP and its PTs                                           |
| <b>Against co-attestation (§4.2)</b>        |                                                                         |
| Leak eOPF secrets                           | Store in protected memory/storage and use SC-resistant crypto libraries |
| Steal <i>eid</i> using enclave              | Only install to pre-measured enclave                                    |
| Guess <i>eid</i>                            | Use very large number                                                   |
| Replay communication                        | Use Random nonces                                                       |
| <b>Against enclave orchestration (§5.1)</b> |                                                                         |
| Modify LC MSRs                              | Trap writes to MSRs                                                     |
| Hide resource usage                         | Trap EADD; prevent TSC MSR changes                                      |
| Hide malware in enclaves                    | Scan initial content and disable changes                                |
| <b>Against side-channel defense (§5.2)</b>  |                                                                         |
| Write to page tables                        | Write-protect using EPT                                                 |
| Disable/modify CAT                          | Trap writes to MSRs                                                     |
| Enable hyper-threading                      | Monitor interrupts on signalled cores                                   |
| Enable LBR/PT                               | Trap corresponding MSRs                                                 |

Figure 4: Table illustrates how eOPF defends against several attacks directed at its framework and protected services.

The platform is securely provisioned by the trusted cloud provider and all secret keys established during provisioning are securely maintained within the system's protected key management system (e.g., a persistent dedicated storage). Every subsequent cryptographic operation is also securely performed in a side-channel resistant manner ensuring the attacker cannot leak keys while they are being used. Moreover, during enclave installation, the *eid* is directly received by eOPF through a secure communication channel with a remote user and securely kept in protected memory. The possibility of the attacker being able to guess the correct *eid* is infinitesimally small ( $2^{-32768}$ ). This is harder than guessing an RSA cryptographic key. Finally, even though the OS can record and replay network packets, all network communication is secured using random nonces to prevent replay attacks.

### 7.2 Analyzing Services Security

**Preventing attacks against enclave orchestration.** After compromising the OS, the attacker can attempt to run arbitrary enclaves by modifying launch control MSRs or hide their enclave resource usage from the cloud provider. Additionally, an attacker might try to hide malware inside enclaves. eOPF prevents all aforementioned attacks.

eOPF ensures that SGX flexible launch control features can only be configured by the cloud provider by intercepting all writes to the launch control MSRs. Hence, even a user that has compromised the OS cannot run enclaves on a machine without obtaining a launch token from the cloud provider. Since eOPF interposes all enclave supervisor and user interactions, it can trivially measure how the enclave is using resources. In particular, it uses RDTSCP to measure CPU



time on enclave entries and traps all instructions that insert or remove enclave pages. To prevent the attacker from keeping malware inside enclaves, eOPF scans enclave contents at load time and prevents subsequent code changes.

**Preventing attacks against side-channel defense.** The attacker can attempt to disable side-channel defenses by modifying entries inside enclave page tables (e.g., reset access bit), modify CAT configuration to disable last-level cache isolation, resume hyper-threading to leverage per-core side-channels, and re-enable LBR/PT to exploit BTB-related side-channels. eOPF protects against all aforementioned attacks (§5.2).

Access and dirty bits are set in enclave page table entries, and the tables are write-protected (using EPT) to prevent additional modifications. To modify CAT partitions, the OS would need to write to CAT-related MSR (using `WRMSR`) and such a write is trapped by eOPF. If the attacker re-enables hyper-threading, they will be caught since the processor core will raise an interrupt that will be intercepted by eOPF. Finally, LBR and PT configurations can only be changed by writing to MSR, and any such attempt is caught by eOPF.

### 7.3 Analyzing eOPF’s TCB

This section analyzes eOPF’s TCB followed by a brief discussion on the impact of attacks on eOPF to a user.

eOPF allows the OS to handle most functionality and only interposes on sensitive interactions (e.g., MSR writes). Hence, from a cloud machine’s perspective, eOPF only marginally increases the TCB, which can be rigorously tested.

On a virtualized cloud machine, eOPF’s complete TCB includes the hypervisor. We find this acceptable for several reasons. In particular, even though hypervisors can be large, the attacker-exploitable interface is typically significantly narrower than monolithic OSs, resulting in fewer discovered vulnerabilities in hypervisor codebases [28, 77]. The exploit of these vulnerabilities can be made significantly more challenging by using memory lockdown and compiler instrumentation to ensure hypervisor code integrity and control-flow integrity, respectively, with a small performance impact [86]. Moreover, eOPF’s TCB can be reduced using hypervisor compartmentalization [77]. In such scenarios, eOPF can execute alongside a tiny security monitor and enforce security invariants while remaining isolated from the large cloud hypervisor.

Finally, since eOPF is external to the enclave and the microcode, it cannot access enclave contents and, during its operation, it is not exposed to enclave secrets. Hence, attacks against eOPF cannot harm the existing SGX guarantees.

## 8 Performance Evaluation

This section describes eOPF’s performance through custom benchmarks and diverse real-world programs.

**Setup.** We evaluated eOPF using SGX desktop and server machines (Fig. 5). Although SGX is deprecated on desktops,

|                   | Desktop         | Server          |
|-------------------|-----------------|-----------------|
| <b>Hardware</b>   |                 |                 |
| CPU model         | i7-8700         | Xeon Gold 6348  |
| CPU sockets       | 1               | 2               |
| Cores × threads   | 6 × 2           | 28 × 2          |
| Clock speed       | 3.20GHz         | 2.60GHz         |
| Cache (L1/L2/LLC) | 64KB/256KB/12MB | 64KB/1.2MB/42MB |
| LLC ways          | 16              | 12              |
| RAM size          | 16GB            | 512GB           |
| EPC size          | 128MB           | 128GB           |
| <b>Software</b>   |                 |                 |
| Linux kernel      | 5.4             | 5.11            |
| SGX SDK           | 2.3             | 2.15            |
| SGX driver        | Legacy 2.6      | DCAP 1.41       |

Figure 5: Machine platforms used for evaluation.

we used the desktop because we observed a large number of enclave exits on it. In particular, the desktop has a smaller EPC which leads to frequent page faults (which cause exits) when running large enclaves [68]. Many of eOPF’s side-channel defenses incur extra costs at exits; hence, the desktop machine allows us to better observe worst-case overheads.

We leveraged two software optimizations to reduce enclave exits, both of which are well-supported by modern systems. First, unless noted otherwise, we used the *exitless* (or *switchless*) system call setting for all experiments. This setting is now widely-supported (e.g., even the relatively basic SGX SDK supports it [17]) and is known to improve performance by avoiding expensive enclave exits using background request handling threads and system call batching. Other work also evaluates SGX enclaves using this option [22, 68]. Second, we configured both machine kernels as *tickless* [15] to reduce enclave exits due to frequent timer interrupts [66].

**Terminology.** eOPF refers to the base framework with all interposition and cloud orchestration features but no runtime side-channel defense. eOPF+PM refers to the system with the paging module enabled for paging side-channel defense. eOPF+CM refers to caching module enabled on a system to prevent cache attacks, while eOPF+BM refer to the system enabled with BPU defenses. Hyper-threading in enclave-running cores is disabled for both CM and BM. When all side-channel defenses are enabled, we refer to the system as eOPF+PM+CM+BM. Our baseline in all experiments was a non-virtualized system.

### 8.1 Microbenchmarks

This section describes our experiments to find the raw cost of eOPF’s enclave interposition and side-channel protection at different events through two benchmarks.

**Enclave event interposition benchmark.** We created a test program that executes 100k barebones enclave entry and exit tests. In the entry test, the application enters the enclave while providing current (pre-entry) time as argument. The enclave measures the time it took to enter and returns to the appli-

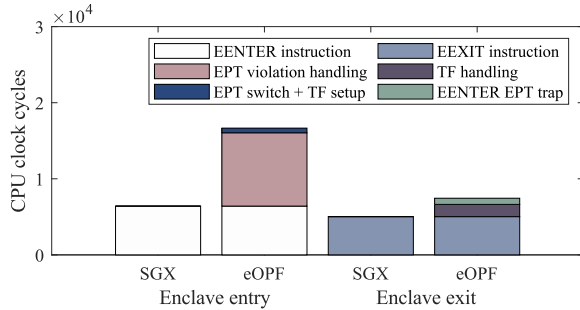


Figure 6: Overheads at enclave entry and exit for eOPF on our server machine. TF means *trap flag*. SGX’s EENTER and EEXIT instruction cost was comparable to existing work [68].

ation. In the exit test, the same set of operations occur but from enclave to the application. To get a detailed picture, we also measured the time to perform eOPF-specific tasks (e.g., switch EPT during entry) from inside the framework. The time was measured everywhere using RDTSCP. Finally, we disabled *exitless* mode for this experiment to get the full cost of exits. Fig. 6 shows the performance overhead incurred in our experiment with native SGX and eOPF.

*eOPF overhead.* eOPF adds 59% overhead to enclave entries, while it adds 71% overhead to enclave exits.

On each entry, eOPF incurs a virtual machine exit to handle the EPT violation, which takes 9639 cycles on our machine. We believe some of this cost is because eOPF is implemented on Bareflank, which is designed for modularity instead of performance; hence, it can be further optimized. Once the violation is handled, eOPF sets the trap flag to trap subsequent exits and switches the EPT to allow enclave execution. These tasks take 1067 cycles.

On each enclave exit, eOPF incurs a virtual machine exit for the trap flag, which only takes 1608 cycles on our machine. Afterwards, eOPF switches the EPT to trap subsequent enclave entries which takes 812 cycles on our machine.

**Side-channel protection benchmark.** We ran a benchmark enclave program that continuously writes to a large 256 MB buffer on both machines. We ran the enclave continuously for 60 seconds and measured incurred performance overheads (using RDTSCP) while enabling different side-channel protection modules. Since the resources affected by the caching and branching modules (CM and BM) have different sizes on each of our test machines, we ran their experiments on each machine. Fig. 7 shows the runtime performance overhead.

*eOPF+PM overhead.* Paging defenses introduce a one-time cost, during the enclave’s lifetime, at enclave creation. The paging module performs the following steps: (a) maps guest page tables to eOPF’s address space, (b) scans entire page tables (including non-enclave entries) to find the enclave regions and sets access/dirty bits, and (c) write-protects enclave page table entries. On the server machine, these steps adds an

| Invalidation     | Time (kcycles) |       | Time (ms) |      |
|------------------|----------------|-------|-----------|------|
|                  | Min            | Max   | Min       | Max  |
| <b>Desktop</b>   |                |       |           |      |
| CM (L1/L2 + LLC) | 247            | 10560 | 0.08      | 3.35 |
| BM (PHT)         | 120            | 844   | 0.04      | 0.30 |
| Total            | 367            | 11404 | 0.12      | 3.65 |
| <b>Server</b>    |                |       |           |      |
| CM (L1/L2 + LLC) | 3240           | 19454 | 1.25      | 7.48 |
| BM (PHT)         | 373            | 494   | 0.14      | 0.19 |
| Total            | 3613           | 19947 | 1.39      | 7.67 |

Figure 7: Overheads due to resource invalidation at enclave exits for CM and BM.

additional  $\sim 1.9$  seconds to our enclave’s creation. We expect this cost is negligible for longer-running enclaves.

*eOPF+CM overhead.* At enclave exits, eOPF’s cache defenses (§5.2.1 and §5.2.2) require (a) partitioning the last-level cache (LLC) and switching partitions during enclave execution and (b) writing back and invalidating the caches at enclave exits. Please refer to §8.2 for the runtime overhead.

Partitioning the LLC and switching partitions is fast: it takes  $\sim 200$  cycles to update a model-specific register (using WRMSR). Cache write-back and invalidation time depends on the state and size of the cache. On the desktop machine, we noticed that it took up to 3.35 ms, whereas its lower bound (through consecutive invalidations) was 0.08 ms. Cache invalidation took from 1.25 ms to 7.48 ms on the server.

Despite a smaller cache, invalidation on the desktop is not that much faster than the server. The reason is that, unlike server machines where SGX does not implement hardware memory integrity [41], the desktop enforces integrity using a Merkle tree. This tree is updated on each cache-line that is flushed to DRAM [46], incurring 6 additional memory accesses per-cache-line. Notably, invalidating non-enclave memory on the desktop machine took only up to 0.81 ms.

*eOPF+BM overhead.* We executed our custom branch predictor flush to invalidate the PHT (§5.2.2). The lower bound for invalidation was 0.04 and we saw an upper bound of 0.30 milliseconds. Since typically branch misprediction adds a 5 nanoseconds latency [36], our evaluation results indicate that all branches were being mispredicted.

**Benchmark result summary.** eOPF interposition and side-channel protection cost is only incurred at enclave entry or exits. Although the cost can be high, these events are only a small fraction of the program’s execution and can be significantly reduced with widely-available optimizations like switchless enclaves and tickless kernels (§8). For instance, in our experiments with SGX SDK’s switchless benchmark [14], we noticed only 3k exits for 2 million enclave calls. Hence, as the next section will demonstrate, eOPF’s overhead on real-world programs using software optimizations is typically modest, even with all side-channel protections enabled.

## 8.2 Real-world Enclave Programs

**Common settings and results.** While partitioning sensitive functionality of a program to run inside an enclave was the initial SGX intent, it has evolved over the years to run entire programs inside enclaves using Library OSs [25, 76, 80, 83], particularly for convenience reasons. In fact, even Intel has officially adopted (and continuously supports) the Gramine Library OS (formerly Graphene [83]) as an SDK for running Linux programs inside SGX [5]. Hence, we also used the Occlum and Gramine Library OSs [76, 83] to run Linux programs inside enclaves for evaluation.

Unless noted otherwise, we ran programs on the server machine using an enclave partition size of  $1/12 \cdot \text{LLC}$ , the smallest allowed CAT-based partition on the server machine. This setting allows the machine to be shared amongst the most number of users, highly desirable in cloud machines. We ran each program 10 times and report the average.

Since eOPF+PM only incurs a one-time performance overhead during enclave creation, in our long-running programs, its performance impact was negligible. Hence, we do not illustrate its overhead in Fig. 8 and Fig. 9

We also evaluated the overhead incurred by the base framework alone (i.e., no side-channel modules were enabled) during each real-world program's execution inside enclaves. The base framework must interpose all enclave events, which increases the runtime cost (§8.1). However, our evaluation shows that this cost is very small during execution. On the server machine, the framework's cost is less than 2% on average during execution of each real-world program described in this section, primarily because exits are low and CPU virtualization (required by the framework) is lightweight.

**Assorted (SPEC).** SPEC is a collection of well-known CPU and memory-intensive programs that are useful to assess real-world system performance. It has been used for evaluation by the Occlum LibOS (which we used for this experiment) and includes real-world programs (e.g., compiler toolchains and compression libraries) that are evaluation targets for other SGX systems [75]. The Occlum LibOS is designed to support SPEC 2006 integer benchmarks out-of-the-box, unlike the latest SPEC 2017. Hence, we decided to evaluate our system with SPEC 2006 integer benchmarks.

Fig. 8 illustrates eOPF's performance across SPEC using reference datasets on the server machine. Encouragingly, even with all protections enabled, most programs incurred a modest performance overhead—7 out of 11 incurred less than 20% slowdown, and the geometric mean slowdown was 17%. Across all programs, the biggest slowdown factor was cache protections. Since our experiments used switchless system calls and tickless kernels (§8), most programs incurred very few enclave exits and showed modest performance overhead. Nevertheless, the smaller enclave LLC partition had a considerable effect (e.g., 311%) on the performance of highly memory-intensive programs like gcc and omnetpp.

We also ran SPEC programs on the desktop machine using test datasets to estimate performance in worst-case scenarios with many enclave exits. Since reference workloads require significant memory, it is infeasible to run them on the desktop machine. Fig. 9 illustrates eOPF's performance on the desktop using  $1/8 \cdot \text{LLC}$ , the closest to fair sharing for each core. With both CM and BM enabled, the geometric mean overhead was 34% on the desktop machine. Since the desktop machine only has a 128 MB EPC, demand paging was inevitable. Thus, the programs incurred many more enclave exits because of page faults and performance was (expectedly) lower than the server machine. We noticed two programs, mcf and sjeng, incurred a very high overhead. We found that their test datasets required up to 1 GB of memory, hence their enclaves incurred the most exits (due to page faults) per-second.

**Key-value store (Redis).** Key-value stores like Redis [12] are widely used in cloud environments. We evaluated Redis using default settings and its official redis-benchmark, which tests 20 different key-value store operations including GET, SET, MSET, and POP. We ran each operation for 100,000 iterations using the default settings of 50 parallel clients. In its default state, Redis only keeps the key-value store *in-memory* for performance and does not write to disk. Additionally, note that while Redis ran inside an enclave, client network socket connections were received by user-space code outside the enclave, since enclaves cannot receive network packets directly. This code then sent the packets to the Redis enclave. In typical scenarios, client request in the packets would be protected using TLS that terminates inside the enclave, but redis-benchmark does not support TLS. Hence, requests were unencrypted and we used this for benchmarking purposes.

In our experiments, eOPF+CM+BM reduced throughput by 4–21% (geometric mean was 11%) across these operations. We only observed 27 enclave exits per-second during the benchmark's 119s execution. These exits were few due to switchless optimizations (§8). Given a low number of enclave exits and the fact that Redis is highly memory-intensive, the major factor behind its throughput reduction was the program executing on a restricted LLC partition.

**Web server (Lighttpd).** Webservers like Lighttpd [8], handle sensitive queries to fetch webpages, and hence are a good fit for SGX. We ran Lighttpd with 8 worker threads because this setting maximized throughput. From a separate server machine (average latency between machines was 0.09 ms), we used ApacheBench to send 10,000 HTTP requests for a 10 KB file from up to 256 concurrent clients. We sent HTTP requests like prior research [76] for stress benchmarking. In real-world cases, HTTPS ensures a request is end-to-end protected with TLS connections terminating inside the enclave.

In our experiments, eOPF+CM+BM's geometric mean throughput reduction across the test was only 5%. Interestingly, requests from a single client incurred a 65% throughput reduction, while requests from 256 concurrent clients incurred only 1% reduction. The reason is that the worker threads go

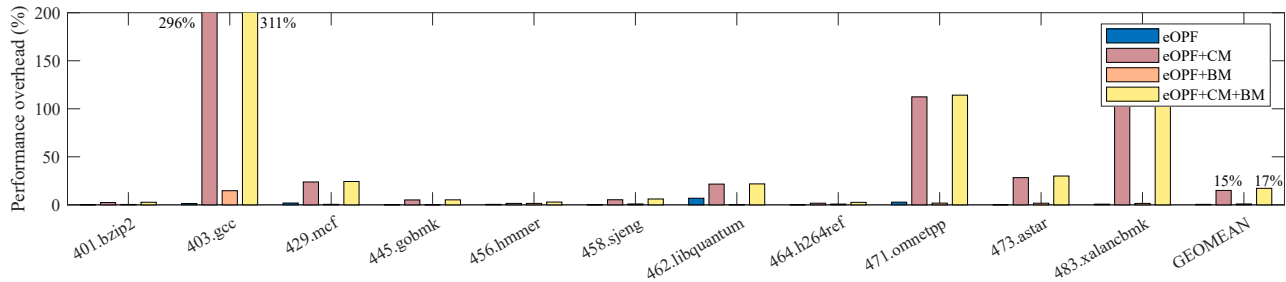


Figure 8: SPEC CPU 2006 performance with eOPF using the reference dataset on the server machine. The enclave partition was  $1/12 * \text{LLC}$ . For this test, the enclave exits per-second were: 3, 105, 4, 3, 3, 2, 2, 1, 11, 1, 8, from left to right.

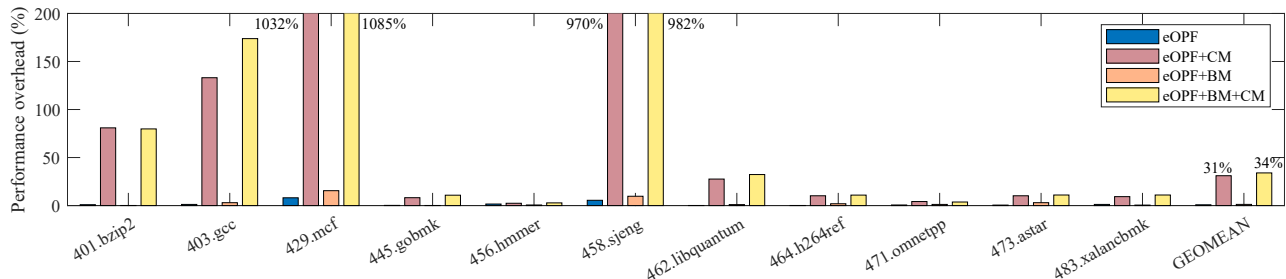


Figure 9: SPEC CPU 2006 performance with eOPF using the test dataset on the desktop machine. The enclave partition was  $1/8 * \text{LLC}$ . For this test, the enclave exits per-second were 29, 2184, 3071, 965, 25, 3444, 417, 298, 60, 22, 91, from left to right.

to sleep when there are no requests and they are awakened through inter-processor-interrupts, hence they incur additional enclave exits. With greater concurrency, the workers are always busy handling requests, thus they do not go to sleep.

### 8.3 Key takeaways

**T1.** The base eOPF (which enables secure enclave orchestration) incurs a low performance overhead ( $< 2\%$ ) on real-world programs because (a) it leverages lightweight techniques and (b) its overhead is incurred at infrequent enclave exits.

**T2.** While eOPF’s overhead expectedly increases with principled side-channel defenses, especially for highly memory-intensive programs (e.g., 311% for gcc on the server machine), it remains modest for the vast majority of programs (e.g., 17% geomean for SPEC programs on the server machine).

**T3.** eOPF’s side-channel defense overhead is comparable to defenses that detect attacks using heuristics (e.g., Varys [66] incurs 15% overhead). However, through invalidation and isolation, eOPF provides strong protection akin to cryptographic techniques that obfuscate *all* side-channel leakage with high costs (e.g., Raccoon [70] incurs  $21.8\times$  overhead).

**T4.** Given the modest defense cost and the fact that eOPF allows users to flexibly decide when defenses are applied, eOPF can be practically adopted in today’s cloud machines.

## 9 Discussion

**Virtual machine support.** In addition to containers, eOPF can orchestrate and protect enclaves running in different vir-

tual machines (VMs) without a design change. This is because eOPF executes at the hypervisor layer, where it has the ability to distinguish between enclaves in different VMs [50, 52] during enclave lifecycle interposition (§4.1). In particular, when the hypervisor starts or resumes a VM, eOPF tracks this using the virtual machine control structure (VMCS). Subsequently, at any exit to the VMM during enclave creation or asynchronous enclave exits, eOPF determines which VM encountered this event by checking the VMCS again. Finally, eOPF interposes on (a) enclave entries using the per-VM EPT and (b) synchronous enclave exits using the single-step interception bit, which is also set in the per-VM VMCS.

**Co-attestation without premeasurement.** While provisioning an enclave with a secret identifier (*eid*) during co-attestation, the requirement is that *eid* is not disclosed to an attacker-controlled enclave (§4.2). In principle, this can be achieved without premeasurement if eOPF (a) installs the *eid* in any recently-created enclave and (b) restricts *eid* enclave page permissions using EPT to prevent the enclave from accessing it, unless SGX measurement is called at which time the user will verify. We leave the study of alternate co-attestation primitives for eOPF to future work.

**Supported enclave count.** By default, eOPF is a thin orchestration layer that collects statistics and enforces properties for cloud providers; hence, it supports as many enclaves as the platform originally can. If *all* side-channel protections are enforced (§5.2), eOPF can support (a) as many enclaves as can be kept resident in the EPC (i.e., within 512 GB in modern systems [41]) and (b) as many *distrusting* containers (each with unlimited enclaves) as CAT partitions allow (currently



15 partitions). The first limit can be removed using oblivious page swapping mechanisms (discussed in the *future eOPF extensions* paragraph in this section). The second limit can be addressed if future hardware iterations increase the number of isolated cache partitions (e.g., using recent proposals [37]). eOPF can be easily extended to leverage new functionality when it is made available by developers or hardware vendors.

**Future eOPF extensions.** eOPF is designed to be an extensible framework that flexibly provides several guarantees to cloud providers and enclave users. One possible extension would be to support *oblivious swapping* of enclave pages at page faults [67]. In particular, when a page fault happens during enclave execution, eOPF can clear the CR2 register to shield the faulting page from the OS. Instead, eOPF can provide a list of candidate pages for the OS to bring into the EPC. To create a secure candidate list, eOPF can rely on a cryptographically-secure algorithm like the Oblivious RAM (ORAM) [82]. eOPF can also enable the use of efficient per-thread hardware memory protection (using MPK) for enclaves and enable memory protection use-cases [49]. In particular, currently enclaves cannot securely use MPK since it requires setting protection keys in page tables [20], which are controlled by the OS. Instead of relying on the OS, the enclave can rely on eOPF to set correct protection keys.

## 10 Related Work

**Privileged software monitors for TEEs.** A lot of research has been done to design software security monitors that are more privileged than the OS and leverage them to create protected process contexts with strong isolation guarantees. Many systems [31, 47, 48, 62] rely on hardware memory protection capabilities (e.g., EPT) of virtualization layers (e.g., VMX) for such security monitors. Other systems [35, 38] rely on compiler instrumentation (e.g., software fault isolation) to deprive the OS and execute a security monitor at ring-0. On non-x86 systems, several designs [34, 41, 42, 59] leverage architectural privileged layers like ARM TrustZone or RISC-V machine mode and their protection features (e.g., physical memory protection). While our design of eOPF takes inspiration from all these systems, eOPF remains unique for several reasons. First, eOPF only offers complementary protection to enclaves, ensuring that even if its monitor is compromised, user computations retain SGX protections. Second, by leveraging SGX and its extensive industry support, eOPF can be readily-adopted by cloud providers without hardware changes or designing extensive software development kits.

**SGX digital side-channel defenses.** Researchers have proposed both software and hardware solutions to address digital side-channels in enclaves. Software solutions implemented inside enclaves cannot prevent memory access patterns from being disclosed since that is a hardware limitation. Therefore, many software protection schemes rely on cryptographic

protocols like ORAM [18, 70, 71] to obfuscate all memory access patterns (i.e., make all access patterns indistinguishable). However, since ORAM is expensive, these defenses incur significant slowdown (e.g.,  $21.8\times$  [70]). Other software solutions [45, 66, 78, 79] leverage heuristics to detect certain attack vectors. On the hardware front, Autarky [67] implements strong and efficient protection against page table attacks. One of Autarky’s ideas is to set all access and dirty bits for enclave page table entries, which is also adopted by eOPF’s page table defense. In contrast to these defenses, eOPF offers a more comprehensive protection against several side-channels with low performance impact and minimal user effort.

**Running programs inside enclaves.** Haven [25] runs Microsoft Windows programs in enclaves with minimal changes. Graphene [83] and Panoply [80] implement library OSs to run Linux applications inside enclaves, while VC3 [73] allows developers to protect data analytics. Ryoan [49] provides trusted client-server application processing in SGX and Scone [22] enables SGX-protected containers. Eleos [68] designs user-level paging to reduce enclave exits and improve performance. eOPF is orthogonal to this line of research and can improve the security guarantees provided by these systems.

**Enclave and platform attestation.** Windows 11 machines leverage the TPM [23] for secure boot and platform attestation. SGX’s remote enclave attestation is also inspired by the TPM. Recently, MAGE [30] demonstrated how to extend SGX enclave to attest mutually-trusted enclaves together by leveraging a *premeasurement* of enclave memory regions. eOPF’s use of premeasurement ( $pM_e$ ) is inspired by MAGE, but eOPF uses it to enable platform-enclave co-attestation.

## 11 Conclusion

eOPF provides a trusted privileged environment for cloud providers to enable protected services on their SGX-capable confidential computing platform. In this paper, we overcome several challenges to design eOPF, implement secure cloud orchestration and complementary side-channel defense as services enabled by eOPF, and provide a detailed security and performance analysis of the framework. Our results indicate that eOPF provides strong protection with very low performance impact on average ( $<2\%$  for the framework alone) and it can be readily-adopted in today’s clouds.

## 12 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Bryan Parno, for their insightful reviews which significantly improved the paper’s evaluation and presentation. This work was partly supported by the National Science Foundation (NSF) under grants CNS-2145888 and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093).

## References

- [1] AWS Lambda. <https://aws.amazon.com/lambda/>.
- [2] AWS Nitro Enclaves. <https://aws.amazon.com/ec2/nitro/nitro-enclaves/>.
- [3] Bareflank/hypervisor. <https://github.com/Bareflank/hypervisor>.
- [4] Intel 3rd Gen Xeon Scalable Processors (Ice Lake). <https://www.storagereview.com/news/intel-3rd-gen-xeon-scalable-processors-ice-lake>.
- [5] Intel(r) Software Guard Extensions. <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html>.
- [6] Key concepts and Definitions for Burstable Performance Instances. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html>.
- [7] L1 Terminal Fault / CVE-2018-3615 , CVE-2018-3620,CVE-2018-3646 / INTEL-SA-00161. <https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault>.
- [8] Lighttpd - Fly Light. <https://www.lighttpd.net/>.
- [9] Nginx. <https://www.nginx.com/>.
- [10] OpenBSD: HyperThreading Disabled by Default on Install. <https://marc.info/?l=openbsd-cvs&m=152943660103446>.
- [11] Qubole Announces Apache Spark on AWS Lambda. <https://www.qubole.com/blog/spark-on-aws-lambda/>.
- [12] Redis. <https://redis.io/>.
- [13] Standard Performance Evaluation Corporation. <https://www.spec.org/cpu2006/>.
- [14] Switchless Enclave Example. <https://github.com/intel/linux-sgx/tree/master/SampleCode/Switchless>.
- [15] Tickless Kernel. [https://en.wikipedia.org/wiki/Tickless\\_kernel](https://en.wikipedia.org/wiki/Tickless_kernel).
- [16] 23andme: DNA Genetic Testing and Analysis, 2017.
- [17] 01org. Intel(r) software guard extensions for linux\* os (source code). <https://github.com/01org/linux-sgx>, 2016.
- [18] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A Commodity Obfuscation Engine for Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.
- [19] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient Multi-client Isolation Under Adversarial Programs. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, 2021.
- [20] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight Data Race Detection with Per-thread Memory Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual Event, USA, April 2021.
- [21] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Taveri. Port Contention for Fun and Profit. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.
- [22] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.
- [23] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.
- [24] Microsoft Azure. Azure confidential computing. <https://azure.microsoft.com/en-us/blog/azure-confidential-computing/>, 2018.
- [25] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.
- [26] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostianen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017.

- [27] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD.
- [28] Ramaswamy Chandramouli, Ramaswamy Chandramouli, Anoop Singhal, Duminda Wijesekera, and Changwei Liu. *Methodology for Enabling Forensic Analysis Using Hypervisor Vulnerabilities Data*. US Department of Commerce, National Institute of Standards and Technology, 2019.
- [29] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
- [30] Guoxing Chen and Yinqian Zhang. MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties. In *Proceedings of the 31st USENIX Security Symposium (Security)*, August 2022.
- [31] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dworkin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [32] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *Proceedings of the 31st USENIX Security Symposium (Security)*, August 2022.
- [33] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
- [34] Victor Costan, Iliia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.
- [35] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from Hostile Operating Systems. *ACM SIGARCH Computer Architecture News*, 2014.
- [36] Jeff Dean. Latency numbers every programmer should know. <https://gist.github.com/jboner/2841832>.
- [37] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stempf. Chunked-cache: On-demand and Scalable Cache Isolation for Security Architectures. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Event, USA, February 2021.
- [38] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug 2018.
- [39] Alan M Dunn, Owen S Hofmann, Brent Waters, and Emmett Witchel. Cloaking Malware with the Trusted Platform Module. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011.
- [40] Dmitry Evtushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [41] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable Memory Protection in the PENGLAI Enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual Event, USA, July 2021.
- [42] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.
- [43] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EUROSEC*, pages 2–1, 2017.
- [44] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug 2018.
- [45] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Security Symposium (Security)*, 2017.

- [46] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204, 2016. <https://eprint.iacr.org/2016/204>.
- [47] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.
- [48] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [49] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.
- [50] Intel. Intel Trusted eXecution Technology—Software Development Guide. *Document number 315168-005*.
- [51] Intel. Intel® processors voltage settings modification advisory. <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html>.
- [52] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide*, 2016.
- [53] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3D: System Programming Guide*, 2022.
- [54] Intel Jason Chen. Supporting TEE on x86 Client Platforms with pKVM. [https://www.youtube.com/watch?v=EP9ps\\_h-WeI](https://www.youtube.com/watch?v=EP9ps_h-WeI).
- [55] Pratheek Karnati. Data-in-use Protection on IBM Cloud using Intel SGX. <https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx>.
- [56] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.
- [57] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*.
- [58] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug 2020.
- [59] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.
- [60] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug 2017.
- [61] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE international symposium on high performance computer architecture (HPCA)*, 2016.
- [62] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, May 2010.
- [63] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, March 2008.
- [64] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on HASP*, 2013.
- [65] Shintarou Okada. a header-file-only, sha256 hash generator in c++. <https://github.com/okdshin/PicoSHA2>.
- [66] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2018.



- [67] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.
- [68] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2016.
- [69] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.
- [70] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.
- [71] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.
- [72] Sarah Schlothauer. Serverless platform Apache OpenWhisk graduates to Top Level Project. <https://jaxenter.com/serverless-openwhisk-top-level-160417.html>, 2019.
- [73] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2015.
- [74] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard eXtension: Using SGX to Conceal Cache Attacks. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2017.
- [75] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.
- [76] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [77] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.
- [78] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.
- [79] S Shinde, ZL Chua, V Narayanan, and P Saxena. Preventing your Faults from Telling your Secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.
- [80] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.
- [81] Splunk. How Good is ClamAV at Detecting Commodity Malware? [https://www.splunk.com/en\\_us/blog/security/how-good-is-clamav-at-detecting-commodity-malware.html](https://www.splunk.com/en_us/blog/security/how-good-is-clamav-at-detecting-commodity-malware.html).
- [82] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.
- [83] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, June 2017.
- [84] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug 2017.

- [85] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, 2019.
- [86] Zhi Wang and Xuxian Jiang. Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland)*, May 2010.
- [87] Richard Wilkins and Brian Richardson. UEFI Secure Boot in Modern Computer Security Solutions, 2013.
- [88] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2015.
- [89] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy (S&P)*.





# Nimble: Rollback Protection for Confidential Cloud Services

Sebastian Angel\*   Aditya Basu<sup>†</sup>   Weidong Cui\*   Trent Jaeger<sup>†</sup>

Stella Lau<sup>‡</sup>   Srinath Setty\*   Sudheesh Singanamalla<sup>◇</sup>

\*Microsoft Research   †Penn State   ‡MIT CSAIL   ◇University of Washington

## Abstract

This paper introduces Nimble, a cloud service that helps applications running in trusted execution environments (TEEs) to detect rollback attacks (i.e., detect whether a data item retrieved from persistent storage is the latest version). To achieve this, Nimble realizes an append-only ledger service by employing a simple state machine running in a TEE in conjunction with a crash fault-tolerant storage service. Nimble then replicates this trusted state machine to ensure the system is available even if a minority of state machines crash. A salient aspect of Nimble is a new reconfiguration protocol that allows a cloud provider to replace the set of nodes running the trusted state machine whenever it wishes—*without* affecting safety. We have formally verified Nimble’s core protocol in Dafny, and have implemented Nimble such that its trusted state machine runs in multiple TEE platforms (Intel SGX and AMD SNP-SEV). Our results show that a deployment of Nimble on machines running in different availability zones can achieve from tens of thousands of requests/sec with an end-to-end latency of under 3.2 ms (based on an in-memory key-value store) to several thousands of requests/sec with a latency of 30ms (based on Azure Table).

## 1 Introduction

Cloud providers today offer *confidential computing* services where VMs support *trusted execution environments* (TEEs) in which a customer’s code is isolated from other code (including the hypervisor). The promise of TEEs is that customers’ applications enjoy security properties even if the provider is compromised, such as confidentiality of the application’s memory, and the integrity of the application’s execution.

Unfortunately, TEEs do not provide persistent state. If a TEE crashes or is maliciously restarted, its volatile state is lost. Applications running in TEEs must therefore explicitly address this. A common approach is for applications to persist their state in cloud storage services and to use cryptographic primitives such as authenticated encryption to protect that state so that it remains confidential and is not modified by a compromised storage service or OS. But encryption alone does not prevent *rollback attacks*. In such attacks, the provider restarts a customer’s TEE; when the application then attempts to recover its volatile state from persistent storage, the provider intercepts the request and returns old data.

Rollback attacks can be terribly harmful. For instance, con-

sider Signal’s “secure value recovery”, which is a service that runs inside TEEs in a public cloud and allows Signal’s users to back up cryptographic keys or other secret data under a short PIN [6]. Users can establish a TLS session with the service in the TEE, provide their PIN, and recover their key. To prevent an attacker from brute forcing a PIN, Signal enforces a quota on the number of times a wrong PIN can be entered by persisting a counter that tracks the number of attempts made so far. But a compromised cloud provider who wishes to brute force the PIN could simply make some guesses, crash the application, rollback the value of the counter, and retry. This example is far from unique; other types of situations where rollback attacks are problematic include financial transactions, revocation of certificates, access control changes, etc.

Given the significance of rollback attacks, we ask: *how can a cloud provider deploy a service that customers’ confidential computing applications can use to detect rollback attacks?* Importantly, since this will be a core service that runs within a cloud provider, it is paramount that any solution be simple to understand and implement, and that the *trusted computing base* (TCB) be small and easy for customers to audit.

**Prior solutions and their limitations.** One can use a replication protocol to address rollback attacks. For example, the client could keep the latest version of their data replicated across a set of machines, and, assuming a threshold number of these machines are honest, the client can obtain the latest state. In confidential computing, a cloud provider can offer a cloud service that runs a BFT protocol inside TEEs. For many prior works [13, 21, 26, 35, 39, 52], this requires only a small amount of trusted code inside TEEs (typically code to manipulate a counter or a log) and they guarantee safety as long as the trusted code is correct. The drawback is that, to our knowledge, these works do not support *reconfigurations* where the set of TEEs changes over time. Meanwhile, reconfiguration is an absolute necessity: a cloud provider needs the ability to replace failed nodes or migrate healthy nodes whenever it wishes to perform maintenance and updates.

An alternate approach pursued by several proposals [6, 41, 49] is to run *an entire replication protocol* inside TEEs. If the original replication protocol supports reconfigurations, then it stands to reason that the resulting system inside TEEs might do so as well. The drawback is that this significantly increases the complexity and size of the TCB. Meanwhile, for a cloud



service, it is crucial that customers be able to audit the code in the TCB, which is not possible given the complexity and nuance of replication protocols.

**Our solution.** In order to simultaneously support efficient and safe reconfigurations while maintaining a small TCB, we depart significantly from prior works with two key observations. First, state machine replication (SMR) protocols are complex because they must guarantee both *safety* (for rollback resistance this means that any value a client reads is the latest value that had been written) and *liveness* (that a client's request is eventually processed). However, we can separate these two concerns and focus the efforts of the TCB on guaranteeing safety. Liveness can be done outside of the TCB. Such a design is acceptable because, in a cloud setting, a compromised provider can trivially violate liveness anyway (by simply refusing to run the client's code in the first place); liveness, therefore, is a property that an *honest* provider implements for its own sake to ensure that its service is good and highly available.

Second, we observe that a lot of the complexity with SMR is already implemented by existing storage services so one should not reinvent or reimplement the wheel. Instead, we should leverage existing services as much as possible to achieve liveness without significant engineering effort.

By leveraging the above observations, we design *Nimble*, a new SMR protocol that features a small and simple TCB for guaranteeing safety *even in the presence of arbitrary reconfigurations*, and a simple untrusted protocol that reuses existing infrastructure to tolerate faults and to keep the system live. Indeed, owing to Nimble's small TCB, we have formally proven the safety of Nimble's core protocol using the IronFleet [27] methodology with the Dafny program verifier [31].

Nimble is architected as a traditional cloud service built on top of a crash fault-tolerant cloud storage service, providing the interface of a highly-available append-only ledger service. When clients write data to Nimble, Nimble appends this data to their ledger. To ensure that the ledger service provides safety even when the provider is *Byzantine*, Nimble runs a small amount of trusted code that we refer to as an *endorser* inside a TEE. The job of the endorser is very simple: it holds the tail of the ledger in its protected volatile memory. When a client asks for the most recent block written to a ledger, the client provides a *nonce* (a cryptographically random value) to the service. The service forwards this nonce to the endorser who then returns a signature of the current ledger's tail and the nonce. The service then gives the client the data in the ledger in addition to the signature from the endorser that establishes the freshness of said data. Nimble cannot rollback a ledger protected by an endorser because endorsers have no API to rollback their state!

Of course, an endorser can crash and lose its volatile state, so Nimble relies on a set of endorsers. Nimble ensures safety by requiring that there be a quorum of signatures for the nonce and the tail. Nimble ensures liveness (under an honest

provider) by instantiating multiple endorsers. A crucial aspect of Nimble's design is that, since the fault-tolerant storage service already establishes a total order of operations, endorsers do not need to run a replication protocol among themselves.

A remaining challenge is that Nimble needs a mechanism to add, replace, or remove endorsers. This protocol must ensure safety even when the provider is fully untrusted, and must not impede progress when the provider is honest. To address this, Nimble includes a novel reconfiguration protocol that preserves these desirable properties.

To demonstrate the simplicity and applicability of Nimble, we have implemented Nimble on top of both Intel SGX and AMD SEV-SNP, as well as several storage services: an in-memory key-value store, a local disk filestore, MongoDB, and Azure Tables. Our implementation of Nimble with the in-memory key-value store can process 50K requests/sec with an end-to-end latency of under 3 ms. Our geo-replicated Azure tables implementation can also process 50K reads/sec with under 3 ms of median latency; write throughput is more modest, at around 3K writes/sec (without any batching), with an end-to-end latency of 30 ms.

We also demonstrate how to port a significant application to use Nimble by equipping the Hadoop distributed file system (HDFS) with rollback protection. With Nimble-HDFS, data analytics applications running in confidential computing can be certain that the data they read from it is the latest version and has not been rolled back.

**Limitations.** One of the key design tenets behind Nimble is simplicity: both from the perspective of customers that must audit the trusted parts of the system but also from the perspective of engineering teams that must implement and deploy Nimble. This is why we chose to reuse existing storage services rather than implementing our own. As a result, Nimble's implementation inherits the performance of existing systems and may in some cases be more costly than alternatives that do not use storage as a black box. For example, a co-design of Nimble's endorser and the fault-tolerant storage system could save a network round trip, but at great engineering expense.

## 2 Context and rollback attacks

This section provides context, and introduces rollback attacks and their effect on various applications.

### 2.1 Context: Confidential computing

Cloud providers such as Google and Microsoft offer *confidential computing* services where customers' applications run on *trusted execution environments* (TEEs) provided by hardware (e.g., Intel's SGX, AMD SEV-SNP). TEEs encrypt and integrity protect the memory of an application or a whole VM. Additionally, when a cloud provider launches a TEE, through a mechanism known as *remote attestation* (discussed below), a customer can verify that their binary is the one executing in the TEE. Confidential computing promises to help customers run high-assurance applications in the cloud, which,

for example, operate on sensitive data that they wish to keep hidden from the cloud provider. Some proposed and deployed applications for confidential computing are key vaults [6], data analytics [1], machine learning [5], data aggregation [7], auctions [8], and contact discovery for messaging apps [37].

**Remote attestation.** A major component in confidential computing is remote attestation, where a client can confirm that the code running in a TEE is indeed the expected code. Details are elsewhere [19], but at a high level, the TEE produces a *quote* (which contains, among other things, the hash of the binary that was loaded into memory) and signs it with an attestation key that is part of the hardware. A client can verify this quote through a variety of ways, including checking a certificate chain or contacting an attestation service.

**Persistent state.** In practice, applications need to persist data in a storage service (e.g., S3, DynamoDB, Cosmos DB) that lives outside of the TEE’s memory for later retrieval (e.g., in case they restart). To preserve the data’s confidentiality and integrity, TEEs can use *authenticated encryption* prior to externalizing any state. Specifically, a TEE can store data encrypted in a database or on disk, and retrieve it at a later time—and check that it is a valid ciphertext to ensure the data has not been modified. In addition to authenticated encryption, TEEs can use other cryptographic techniques such as oblivious RAM [25] to ensure that their data access patterns are additionally kept private.

**Physical attacks.** Modern TEEs such as the latest version of Intel SGX, Intel TDX, AMD SEV-SNP, and AWS’s Nitro cannot protect against attackers who have physical access to the TEE (they give up memory integrity properties in favor of higher performance). However, existing confidential computing applications already accept this threat model, as otherwise they would not be running inside the TEEs of cloud providers. As a result, a solution to rollback attacks need not defend against physical attacks either.

**Side channel attacks.** There is a large literature [15, 17, 30, 38, 47, 48, 51] that has identified software attacks that can extract information kept in TEE’s memory or violate the integrity guarantees provided by TEEs. Nevertheless, hardware vendors have in the past promptly patched vulnerabilities when discovered and reported by researchers, and there are academic hardware designs that are provably safe from many types of side channels [20]. We believe that TEEs will be more robust to these types of attacks over time.

In this work, we consider these attacks to be out of scope. As we discuss next, rollback attacks are challenging enough even in the absence of these other orthogonal issues.

**Rollback attacks.** While authenticated encryption provides ciphertext integrity and plaintext confidentiality, it does not ensure that the data is *fresh*. In particular, a malicious storage service could provide a valid ciphertext (encrypted and signed by the TEE), that is not the latest version. In Section 1 we discuss Signal’s “secure value recovery” and showcase how

it can be subverted by an attacker who rolls back the application’s state. This same class of attacks can be performed against a banking application (e.g., rolling back a payment), a confidential VM (rolling back to an older version of the OS that has a vulnerability), etc. It is therefore imperative that confidential computing environments have a way for applications to detect such attacks.

### 3 Rollback protection

This section describes a solution for applications running inside TEEs to detect rollback attacks.

**Characterizing rollback attacks.** An application running inside a TEE experiences a rollback attack when one of the following events happens: (1) *stale responses*, where a malicious storage service provider returns a prior version of data instead of the latest (i.e., lack of freshness), possibly because the malicious storage service forks the views of its clients [33]; (2) *synthesized requests*, where a malicious provider synthesizes requests on its own (i.e., they were never issued by the application) and applies them to the storage (thereby affecting future reads); or (3) *replay*, where a malicious provider uses valid requests that were previously sent by the application and applies them to the storage again.

#### 3.1 Our solution

**Addressing stale responses.** It is well known that *linearizability* [28], a widely studied correctness criterion, captures freshness. Informally, a system satisfies linearizability if every operation on an object in the system appears to take place atomically, in an order that is consistent with the real-time ordering of the operations themselves. For example, if an operation  $W$  completes before another operation  $R$  begins, then  $R$  must observe the effects of  $W$  and complete after it.

Our solution to address stale responses is to rely on an append-only ledger service that guarantees linearizability [28] even when the provider is compromised (in Sections 4 and 5, we describe a novel instantiation of such a ledger service, with high performance and a small TCB). In particular, whenever the application wishes to update its persistent state, it stores its updated state in a *block* and appends the block to the ledger; whenever the application wishes to read back its persistent state, it reads the block found at the ledger’s tail.

**Addressing synthesized requests.** We require an application running in a TEE to hold a signing key in a signature scheme that is known only to the application. When the application stores its state in the aforementioned ledger, the application first signs the state with its signing key and then stores the state and the signature in a block. When reading its state from a ledger, the application verifies that there is a valid signature.

**Addressing replay.** To prevent a malicious provider from replaying prior appends, we make the following modification to the solution described thus far: the signature stored in an appended block covers not only the application’s state, but also

the position of the block in the ledger. When reading its state from a ledger, an application verifies that the returned position of the tail matches the position covered by the signature.

Assume that the application's configuration includes a label of the ledger (we denote this as  $\ell$ ). Let  $(sk, vk)$  denote the application's signing and verification keys in a signature scheme, with `Sign` and `Verify` methods. The application maintains a counter, which we denote with  $c$  (when the application is launched for the first time, it can execute the read protocol to set  $c$ ).

**Update protocol.** When an application wishes to update its persistent state from  $S$  to  $S'$ , it does the following:

- Issue `append( $\ell, B, c + 1$ )` and get `receipt`, where  $B = (S', \sigma)$ , and  $\sigma = \text{Sign}(sk, (\ell, S', c + 1))$ .
- If the `append` succeeds and `receipt` is valid, update  $c \leftarrow c + 1$ , else follow the steps in the read protocol.

**Read protocol.** When an application wishes to retrieve persistent state, it does the following:

- $nonce \leftarrow \text{random}()$  // e.g., 128 bits
- $(i, B, receipt) \leftarrow \text{read\_latest}(\ell, nonce)$
- Parse  $B$  as  $(S, \sigma)$ .
- If `Verify( $vk, (\ell, S, i), \sigma$ )` passes and `receipt` is valid with respect to `nonce`, set  $c$  to  $i$  and use  $S$  as the state. If not, abort with `Err("rollback detected")`.

**Putting things together.** We require an append-only ledger service with the following intuitive APIs. We provide a concrete instantiation of such a service in Sections 4 and 5.

- `new_ledger(label) → (ack/nack, receipt)`
- `append(label, block, exp_index) → (ack/nack, receipt)`
- `read_latest(label, nonce) → (index, block, receipt)`

A receipt is a cryptographic object that the client uses to verify that the operation was executed correctly. The nonce in `read_latest` is a random value that prevents the service from caching and returning old receipts, since each receipt must cover the nonce. With these checks, a client obtains linearizability [28] from the ledger service.

The expected index (`exp_index`) in `append` is a directive provided by the application to the ledger service. Its purpose is to help the *honest* ledger service determine whether it can store a particular block in the current tail or whether it must reject the request and notify the application (an honest ledger service just needs to maintain for each ledger how many entries are already appended to support these semantics). In other words, it acts as a type of concurrency control. This is important when the application is concurrent and one of its threads has already stored a block at that position in the ledger. Given the signature in the block, the ledger service cannot append a block anywhere different than its expected index, as the client would detect the inconsistency when it

verifies the signature.

Observe that in the read protocol, if the client's check passes, the service could not have rolled back state. By assumption, a receipt is valid for a random nonce if and only if the service is linearizable. So, it cannot lie about the index  $i$  in the response (i.e., the number of entries in the ledger). Furthermore, in the update protocol, the client embeds crucial metadata (including the expected index of the tail) and signs it with its private key. Since the service provider cannot forge signatures, it cannot return data that was not previously stored at that index by the client.

### 3.2 Storing state in an existing storage service

While the ledger service can be used to append arbitrary data, it has a very limited API. Therefore, it is often better for an application to store their state in some existing (untrusted) storage service better suited to its needs. For example, use a storage service that has better performance for large data, or one that supports things like random access reads and writes, scans, search, stored procedures, etc. We now extend the solution from the prior subsection to support this.

In a nutshell, for updates, the application proceeds in two steps: (1) it persists its state in an existing storage service and then (2) stores a cryptographic digest of that state in the ledger service using the Update procedure described above. Similarly for reads, the application reads state from the storage service and a digest from the ledger service, and in addition to the checks described thus far, it checks that the digest of the state retrieved from the storage service equals the digest from the ledger service. There is one key issue that does not affect safety, but affects liveness: the application may fail after it performs step (1) but before step (2), during updates.

We address this as follows. During update, instead of only storing  $S'$  in the storage service, the application stores  $(S', c + 1, \sigma)$ . This ensures that, by design, the storage service holds a counter that is at most one higher than the counter stored on the ledger service (the storage service cannot tamper with  $S'$  or  $c + 1$  since they are protected by  $\sigma$ ). When an application restarts, it can check if the counter in the ledger service is one lower than the counter in the storage service. If so, this implies a failure after the application updated the storage service but before it updated the ledger service. Therefore, the application uses  $S', c + 1$ , and  $\sigma$  from the storage service to complete its pending append to the ledger service.

Note that the above mechanism will not lead to the corruption of the state in the ledger even if the client mistakenly performs the update more than once. For example, suppose that the client first issues its update operation to the ledger and then fails. When the client restarts, the ledger has not yet processed the update operation (perhaps it is sitting in a queue somewhere) so the client receives the old counter ( $c$ ) from the ledger. If the client re-issues the update, at most one of the two requests (the one in the queue or the freshly issued one) will be applied since they have the same expected index.



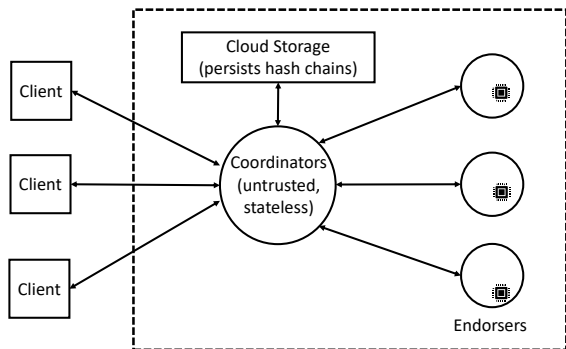


FIGURE 1—Nimble’s architecture (see text for details).

**Concurrency.** If the application has multiple processes that operate on the same persistent state, then we require the storage service used by the application to be linearizable. Furthermore, the application must be able to deal with the failure of its processes and achieve exactly-once semantics [42, 54].

## 4 An overview of Nimble

This section describes Nimble, an append-only ledger service that fills the key role in our rollback protection method (§3).

**Design goals.** To support its target application of rollback protection for confidential services, a key guarantee that we desire from Nimble is linearizability [28]. Unfortunately, an untrusted service can trivially violate linearizability by returning stale responses or by presenting different views of a ledger to different clients [33, 36]. As we discuss below, Nimble will make such violations detectable by relying on certain operations being performed correctly inside TEEs.

Given Nimble’s reliance on TEEs, our second design goal is to naturally ensure that this trusted code is as small as possible and simple enough that it can be audited by customers.

If the cloud infrastructure on which Nimble is hosted is malicious, it can trivially violate liveness (by literally not running the client’s application in the first place), so our third goal is to ensure that if an honest provider runs Nimble as specified, the service will be live.

Finally, we wish to avoid reimplementing complex replication protocols: optimizing them, and ensuring that they are correct and comply with a plethora of business and technology standards for deployment within a public cloud is hard and time consuming. So, we wish to leverage existing services.

**Threat model and assumptions.** Nimble assumes that TEEs work as intended: they protect the memory and the execution of any application running within it from attacks (§2). Nimble’s code running outside TEEs is untrusted by clients, and may behave arbitrarily. Nimble makes standard cryptographic hardness assumptions for its safety guarantee. Nimble ensures liveness during sufficiently long periods of synchrony and when the service follows its prescribed protocol.

As we discuss below, providing safety requires Nimble

to run a trusted state machine, called an endorser, inside a TEE. Since TEEs can crash and lose their state, Nimble runs a collection of endorsers. Unfortunately, if Nimble loses a majority of its endorsers, it must either give up safety or liveness. We discuss this further in Section 9.1. Additionally, for now, we assume that an endorser’s code does not change over time. We discuss possible solutions to this in Section 9.2.

**Design and architecture.** Figure 1 depicts Nimble’s architecture, which is analogous to that of a traditional cloud service. Nimble employs a collection of worker processes, which we refer to as *coordinators*. They are stateless and untrusted, and their job is to process requests from *clients* (i.e., customer applications running in TEEs). For each ledger, Nimble maintains a *hash chain* (a linked list where each node contains data and a cryptographic hash of the previous node) and stores that hash chain in an existing untrusted cloud storage service (e.g., Azure Table). Note that this storage service is completely separate from (and may even be different from) the storage service used by clients to store their data (§3.2).

To guarantee linearizability despite using an untrusted cloud storage, Nimble runs a trusted state machine inside a TEE, which we refer to as an *endorser*. An endorser stores, for each ledger, the tail of the associated hash chain in its memory. An endorser’s code is public, and when launched inside a TEE, it produces a fresh key pair for a signature scheme such that the TEE platform can prove (via remote attestation) that the public key belongs to a legitimate endorser. An endorser uses its secret key to sign its response to any append or read operation. Since an endorser’s state is volatile, Nimble runs a collection of endorsers to achieve fault tolerance. When Nimble boots up, it produces a unique and static identifier that is derived by hashing the public keys of the endorsers. We assume that this identifier is public knowledge.

For each request issued, a client expects a response and a receipt. A receipt contains a list of public keys and signatures that cover the response and the public identifier. The client first verifies that the public keys in the receipt belong to legitimate endorsers based on remote attestation.<sup>1</sup> The client then verifies that there is a quorum of valid signatures from the public keys in the list (in Nimble, a quorum is a majority of endorsers). This is analogous to *witness cosigning* [46]. Additionally, for *read\_latest*, a client sends a nonce and checks that the endorsers’ signed message includes the nonce. This prevents a malicious service from replaying a stale receipt.

To support *reconfigurations* (i.e., addition and/or removal of endorsers), an endorser maintains additional bookkeeping and performs additional checks when the set of endorsers changes. Similarly, a coordinator maintains additional state in the cloud storage service and implements an untrusted distributed protocol. Details are in the next section.

<sup>1</sup>Clients can cache public keys to avoid verifying that they belong to legitimate endorsers. So, clients only need to do remote attestation when the endorser set changes or they lose their local state.



**Small TCB.** Nimble achieves a small TCB because endorsers contain only safety-critical aspects of a replication protocol. For example, to process appends and reads, endorsers maintain tails of hash chains and provide signed responses to requests. To support reconfigurations, they maintain additional state and perform checks but that code is only concerned with safety but not liveness.

## 5 Design details and correctness

This section describes the details of Nimble’s design. We begin with a core protocol that does not support reconfigurations and then describe modifications to support reconfigurations.

### 5.1 Core protocol

Nimble is a replicated state machine with a new architecture where its APIs (see Section 3) are first built on top of an existing crash fault-tolerant storage service, which we refer to as the *untrusted state machine*. A stateless coordinator process can use the untrusted state machine as a black box to implement Nimble’s APIs. To obtain linearizability in the presence of malicious behavior, Nimble uses a collection of endorsers running another state machine, which we refer to as the *endorser state machine*, inside a TEE (a minority of endorsers in the collection can crash).

We first describe the state machine run by endorsers and the untrusted state machine, and then describe how they are invoked in the end-to-end protocol.

**Endorser’s state machine.** Let  $(\text{KeyGen}, \text{Sign}, \text{Verify})$  denote a signature scheme and  $\mathfrak{H}$  a collision-resistant hash function. In our implementation, each state transition of an endorser is atomic (achieved via synchronization primitives) and an endorser state machine provides linearizability.

#### Endorser’s state

- $sk$ , a secret key in a digital signature scheme.
- $\text{status} \in \{\text{“uninitialized”}, \text{“active”}\}$ .
- $id$ , the identity of a particular instance of Nimble.
- $M$ , a label-value map, where a label is a byte vector and a value is tuple  $(t, h)$  in which  $t$  is the tail node of a hash chain and  $h$  is an unsigned integer specifying the number of entries in the hash chain.

**Untrusted state machine.** As noted earlier, Nimble relies on a crash fault-tolerant storage service. In particular, Nimble uses this storage service to realize the untrusted state machine. It does so by storing all the necessary state in the storage service, and using its standard APIs (e.g., put, get, insert, conditional update, atomic batch update) to carefully mutate the state to achieve the desired semantics.

The untrusted state machine is same as the endorser’s state machine, with two key differences. First, it does not generate a key pair nor does it provide a TEE attestation (naturally, it does not sign any of its responses). Second, it stores all entries

appended to a ledger not just the tail entry and provides an API to access ledger entries by their index (as we see below, this is useful for providing liveness in certain cases).

#### Endorser’s state transitions

```

1: fn bootstrap
2:    $(sk, pk) \leftarrow \text{KeyGen}(1^\lambda)$ 
3:    $\text{status} \leftarrow \text{“uninitialized”}$ 
4:   return  $(pk, a)$  //  $a$  is a TEE attestation proving that  $pk$  belongs to a legitimate endorser running inside a TEE.

5: fn initialize( $c$ )
6:   if  $\text{status} \neq \text{“uninitialized”}$  then return  $\text{Err}(\text{AlreadyInit})$ 
7:   if  $pk \notin c$  then return  $\text{Err}(\text{NotInConfig})$ 
8:    $id \leftarrow \mathfrak{H}(c), M \leftarrow \perp, \text{status} \leftarrow \text{“active”}$ 

9: fn new_ledger( $\ell$ )
10:  if  $\text{status} \neq \text{“active”}$  then return  $\text{Err}(\text{NotInit})$ 
11:  if  $\ell \in M$  then return  $\text{Err}(\text{LedgerExists})$ 
12:   $M.\text{insert}(\ell, (0, 0))$ 
13:  return  $\text{Sign}(sk, (\text{“new\_ledger”}, id, \ell, 0, 0))$ 

14: fn append( $\ell, b, \text{exp\_index}$ )
15:  if  $\text{status} \neq \text{“active”}$  then return  $\text{Err}(\text{NotInit})$ 
16:  if  $\ell \notin M$  then return  $\text{Err}(\text{LedgerDoesNotExist})$ 
17:   $(t_{\text{prev}}, h_{\text{prev}}) \leftarrow M.\text{get}(\ell)$ 
18:  if  $\text{exp\_index} \neq h_{\text{prev}} + 1$  then
19:    return  $\text{Err}(\text{OutOfOrder}, h_{\text{prev}})$ 
20:   $t_{\text{curr}} \leftarrow (\mathfrak{H}(t_{\text{prev}}), b); h_{\text{curr}} \leftarrow h_{\text{prev}} + 1$ 
21:   $M.\text{update}(\ell, (t_{\text{curr}}, h_{\text{curr}}))$ 
22:  return  $\text{Sign}(sk, (\text{“append”}, id, \ell, t_{\text{curr}}, h_{\text{curr}}))$ 

23: fn read_latest( $\ell, n$ )
24:  if  $\text{status} \neq \text{“active”}$  then return  $\text{Err}(\text{NotInit})$ 
25:  if  $\ell \notin M$  then return  $\text{Err}(\text{LedgerDoesNotExist})$ 
26:   $(t_{\text{curr}}, h_{\text{curr}}) \leftarrow M.\text{get}(\ell)$ 
27:  return  $\text{Sign}(sk, (\text{“read\_latest”}, id, \ell, t_{\text{curr}}, h_{\text{curr}}, n))$ 

28: fn append_with_read_latest( $\ell, b, \text{exp\_index}, n$ )
29:  return  $(\text{append}(\ell, b, \text{exp\_index}), \text{read\_latest}(\ell, n))$ 

```

**Coordinator’s workflow.** A coordinator invokes the APIs provided by the endorser state machine and the untrusted state machine to provide the APIs we describe in Section 3.

To initialize a Nimble instance, a coordinator process calls bootstrap on a configurable number of endorsers (denote it as  $n$ ) to obtain their public keys. Let  $c$  denotes the list of public keys and the public identity of the instance is  $\mathfrak{H}(c)$ . The coordinator then calls initialize( $c$ ) on the untrusted state machine and when that succeeds, it calls initialize( $c$ ) on the endorser state machine, waiting for  $\lfloor n/2 \rfloor + 1$  out of  $n$  endorsers to respond. At this point, the system is setup to process requests.

When a client issues a request (e.g., new\_ledger, append, or read\_latest), the coordinator uses the provided argument to call the corresponding API on the untrusted state machine and when that returns, it calls the same API on the endorser state machine with the same argument, waiting for  $\lfloor n/2 \rfloor + 1$  out of  $n$  endorsers to respond. The coordinator collects a quorum of those responses and sends it to the client.

The coordinator retries its workflow to account for network failures (e.g., reordered or dropped messages). Furthermore, a coordinator uses the “OutOfOrder” error returned by an endorser to detect if the endorser is “lagging behind”, and if so, it issues appends to roll forward the endorser’s state using blocks in the untrusted state machine.

**Achieving liveness.** For liveness, we require certain extensions to ensure that a coordinator can eventually produce a valid receipt for each request it successfully executes. Recall that Nimble’s liveness holds only when the provider is honest, so the discussion below is limited to that case.

**new\_ledger and append.** An endorser may execute a request (e.g., an append), but when it responds with its signed message to a coordinator, the network may drop the message. To address this, an untrusted process colocated with the endorser stores the signatures generated by an endorser and provides APIs for a coordinator to retrieve them (ensuring that a coordinator that repeatedly retries can obtain signed messages that were generated). Since this mechanism is needed only for liveness, it does not affect the endorser state machine.

Furthermore, when a coordinator assembles a receipt (recall from Section 4 that a receipt is a quorum of signatures), it persists the receipt in the untrusted state machine alongside an appropriate ledger entry. Once a receipt is persisted, the coordinator calls an API on the untrusted process colocated with the endorser to garbage collect the associated signed message. In this way, even if a coordinator crashes or the message from Nimble to a client is dropped, a client can eventually retrieve a receipt for a new\_ledger or an append request from Nimble.

**read\_latest.** There are cases where a coordinator may not succeed in obtaining a valid receipt for a read\_latest request. Specifically, the coordinator may struggle to obtain a quorum of matching responses as each endorser may be in a slightly different state. This can occur during periods of concurrent appends when one append has been applied in one endorser but not yet in another. Nimble addresses this as follows.

Suppose that a coordinator thread (say “read thread”, for ease of reference) is unable to obtain a valid receipt for a read\_latest request. After a configurable number of retries, the “read thread” persists the pending read\_latest request in the untrusted state machine. Furthermore, we modify the coordinator’s workflow so that when a coordinator’s thread receives an append (call it an “append thread”), if there is a pending read\_latest, it invokes the **append\_with\_read\_latest** API of endorsers to execute both the append and the pending read\_latest as an atomic operation. The “append thread” also persists receipts from the endorsers in the untrusted state machine, including the receipt of the pending read\_latest. Meanwhile, the “read thread” retries invoking read\_latest on endorsers and polls the untrusted state machine to see if a concurrent thread produced a receipt via the **append\_with\_read\_latest** API; one of these code paths will eventually succeed.

**Safety and liveness guarantees.** We have a formal specification and a proof of safety in Dafny [31] (our proof uses

IronFleet’s state machine refinement technique [27]). A challenge in our context is that we must account for arbitrary responses from a coordinator and the storage service whereas IronFleet only considers crash faults.

**Lemma 5.1.** *Assuming the integrity and confidentiality guarantees provided by TEEs for executing the specified endorser state machine and standard cryptographic hardness assumptions, whenever Nimble produces a valid receipt for a request, Nimble respects linearizability.*

*Proof (sketch).* By design and implementation, and the stated assumptions, each endorser’s state machine is linearizable. Now suppose that Nimble produces valid receipts for two requests  $R_1$  and  $R_2$ . Suppose that the quorum of endorsers that produce a valid receipt for  $R_1$  and  $R_2$  are respectively  $Q_1$  and  $Q_2$ . In Nimble, a quorum size is a majority, so  $Q_1 \cap Q_2$  must have at least one endorser. Let  $e \in Q_1 \cap Q_2$ . Given that  $e$ ’s state machine is linearizable, then it follows that it processed requests  $R_1$  and  $R_2$  such that it respects linearizability. Since a valid receipt requires that a quorum of endorsers sign the same response, Nimble’s response must equal  $e$ ’s response. This implies that Nimble is linearizable.  $\square$

**Lemma 5.2.** *When the service is honest and during sufficiently long periods of synchrony, and when clients submit requests that can be successfully executed, a coordinator process can eventually generate valid receipts.*

*Proof (sketch).* The claim holds for new\_ledger and append requests because they are first performed on the untrusted state machine and the same operation is applied on a quorum of endorsers in the same order. Even if coordinators are subject to crash failures, or the network drops or reorders packets, the untrusted state machine is linearizable and internally fault tolerant by design and implementation. Since the endorser’s append API takes an additional hint, expected\_index argument, this allows an honest coordinator to apply appends in an endorser in the same order as it is applied in the untrusted state machine. Furthermore, if the network drops signed messages from an endorser to the coordinator, the coordinator can eventually retrieve them. Thus, a coordinator can eventually obtain a valid receipt for new\_ledger and append requests.

For read\_latest requests, there are two cases: (1) no concurrent appends, and (2) concurrent appends. In the first case, a coordinator eventually succeeds in obtaining a quorum of signed messages on the same value (if an endorser is lagging behind, the coordinator can provide missing appends from the untrusted state machine to bring them up to date). In the second case, a coordinator uses append\_with\_read\_latest to combine the read\_latest request with a concurrent append request to obtain a valid receipt.  $\square$

## 5.2 A safe and live replacement of endorsers

Nimble’s core protocol is restricted to a static collection of endorsers. Unfortunately, this restriction is unrealistic: once

an endorser fails, the system loses its initial level of fault-tolerance. Furthermore, once a majority of endorsers fail, the system is permanently unavailable as it cannot process requests nor produce valid receipts ever again.

Nimble includes mechanisms to introduce new endorsers, and to retire existing endorsers, thereby enabling it to proactively maintain a sufficient number of endorsers and avoid the aforementioned issues.

A core challenge is that a proactive replacement of endorsers must not allow untrusted components in the system (e.g., a coordinator) to abuse it to violate linearizability. Another challenge is that the system should not enter a deadlock state if a coordinator process performing such a replacement fails at an inopportune time. In our context, there is another challenge: to achieve high performance, there is no total ordering of all operations. That is, in Nimble’s core protocol, operations on different ledgers proceed in parallel and they only need to be processed by a majority of endorsers (i.e., not all endorsers). Thus, at any point in time, the state of endorsers might not be identical.

Unfortunately, existing solutions are a poor fit in our context. If Nimble were to adopt the folklore solution of keeping membership (i.e., the identity of “current” endorsers) as part of state that is replicated, the system will necessarily have to impose a total order on all operations. This adversely affects performance. Alternatively, each ledger could maintain its own membership state (i.e., each ledger has the list of endorsers that are responsible for endorsing operations on that ledger). However, to change endorsers, the system will need to invoke  $N$  instances of the view-change protocol, where  $N$  is the number of ledgers. In practice,  $N$  can be millions or more, so it is not practical.

Below, we describe how Nimble addresses these challenges *without* bloating the TCB. In a nutshell, Nimble’s solution can be viewed as a way to resolve the tension between the two existing solutions described above.

**Nimble’s reconfiguration protocol.** We introduce a new safe and live protocol, orchestrated by a coordinator, to proactively replace endorsers. At a high level, Nimble proceeds in a sequence of configurations, where in each configuration, a particular set of endorsers, identified by their public keys, are responsible for producing receipts. Nimble’s endorsers keep track of the current configuration as well as the immediately preceding configuration (we denote these with  $C_{curr}$  and  $C_{prev}$  in the endorser’s state machine). Furthermore, when an endorser joins a configuration it “takes over” a previously generated public identity of a Nimble instance only when it can confirm that a quorum of endorsers of the prior configuration have “renounced” it. Moreover, every response signed by an endorser covers, in addition to the public identity, their value of  $C_{curr}$ . This ensures that responses produced by an endorser are tied to a particular configuration and clients can

use it to discover which public keys and quorum size they need to use to verify the responses produced by Nimble.

To switch from one configuration to the next, Nimble’s coordinator proceeds in three phases (this protocol can be invoked at any time). Before we describe these three phases, we provide some preliminaries.

Let  $\mathcal{E}$  and  $\mathcal{N}$  denote sets of existing and new endorsers respectively. In Nimble,  $\mathcal{E} \cap \mathcal{N}$  is an empty set (this simplifies the protocol and proofs, as we discuss in [11, Appendix A]). A coordinator launches endorsers in the set  $\mathcal{N}$ , calls their bootstrap method to obtain their public keys. Let  $C_{\mathcal{E}}$  and  $C_{\mathcal{N}}$  respectively denote the sequence of public keys of endorsers in  $\mathcal{E}$  and  $\mathcal{N}$ . Let  $id$  denote the identity of the Nimble instance (the hash of the list of public keys of the first set of endorsers that bootstrapped the system and that the provider advertises).

At every step in the protocol below, the coordinator reliably persists responses it has received in the untrusted state machine, and by design every step is idempotent. A convenient way to log this information is to use an append-only ledger in the untrusted state machine (an endorser is not explicitly aware of this ledger). Before starting the protocol below, the coordinator appends  $C_{\mathcal{N}}$  to this ledger. Furthermore, all messages logged by the coordinator during the protocol are stored alongside  $C_{\mathcal{N}}$ . As a result, if a coordinator fails or is slow at any point in the protocol, another coordinator can safely take over and complete the remaining steps.

The additional state and transitions needed to support Nimble’s reconfiguration protocol are given below.

#### Endorser’s additional state

- status  $\in$  {“uninitialized”, “initialized”, “active”, “finalized”}.
- $C_{prev}$ , endorsers’ public keys of the previous configuration.
- $C_{curr}$ , endorsers’ public keys of the current configuration.
- $C_{next}$ , endorsers’ public keys of the next configuration.
- $\sigma$ , a signature on the finalized state and other metadata

#### Endorser’s updated initialize function

```

1: fn initialize( $i, m, C_{prev}, C_{curr}$ )
2:   if status = “initialized” then
3:     return Sign( $sk, \langle$ “initialize”,  $id, C_{prev}, C_{curr}, M\rangle$ )
4:   if status  $\neq$  “uninitialized” then return Err(AlreadyInit)
5:   if SecretToPublic( $sk$ )  $\notin$   $C_{curr}$  then
6:     return Err(NotInConfig)
7:   if  $i = H(C_{curr})$  and ( $m \neq \perp$  or  $C_{prev} \neq \perp$ ) then
8:     return Err(InvalidInit)
9:   else if  $i \neq H(C_{curr})$  and ( $m = \perp$  or  $C_{prev} = \perp$ ) then
10:    return Err(InvalidReconf)
11:   $id \leftarrow i, M \leftarrow m, C_{prev} \leftarrow C_{prev}, C_{curr} \leftarrow C_{curr}$ 
12:  status  $\leftarrow$  “initialized”
13:  return Sign( $sk, \langle$ “initialize”,  $id, C_{prev}, C_{curr}, M\rangle$ )

```

### Endorser's additional state transitions

```

1: fn finalize( $C_{next}$ )
2:   if status = "finalized" then return ( $M, \sigma$ )
3:   if status  $\neq$  "active" then return Err(NotActive)
4:    $C_{next} \leftarrow C_{next}, \text{status} \leftarrow \text{"finalized"}$ 
5:    $\sigma \leftarrow \text{Sign}(sk, \langle \text{"finalize"}, id, C_{curr}, C_{next}, M \rangle)$ 
6:    $sk \leftarrow \perp$  // erase the endorser's signing key
7:   return ( $M, \sigma$ )

8: fn activate( $R_{exist}, A, R_{next}$ )
9:   if status  $\neq$  "initialized" then return Err(NotInit)
10:   $q \leftarrow \lfloor |C_{prev}|/2 \rfloor + 1, \kappa \leftarrow \lfloor |C_{curr}|/2 \rfloor + 1$ 
11:  if  $|R_{exist}| < q$  or  $|R_{next}| < \kappa$  then
12:    return Err(InsufficientQuorum)
13:  // Parse  $R_{exist}$  and  $R_{next}$ 
14:   $[(p_1, M_1, \sigma_1, a_1), \dots, (p_q, M_q, \sigma_q, a_q)] \leftarrow R_{exist}$ 
15:   $[(\rho_1, \eta_1, \alpha_1), \dots, (\rho_\kappa, \eta_\kappa, \alpha_\kappa)] \leftarrow R_{next}$ 
16:  for all  $i \in [q]$  do
17:    if  $a_i$  does not attest to  $p_i$  then
18:      return Err(InvalidAttestation)
19:    if  $\neg \text{Verify}(p_i, \sigma_i, \langle \text{"finalize"}, id, C_{prev}, C_{curr}, M_i \rangle)$  then
20:      return Err(InvalidFinalize)
21:  for all  $i \in [\kappa]$  do
22:    if  $\alpha_i$  does not attest to  $\rho_i$  then
23:      return Err(InvalidAttestation)
24:    if  $\neg \text{Verify}(\rho_i, \eta_i, \langle \text{"initialize"}, id, C_{prev}, C_{curr}, M \rangle)$  then
25:      return Err(InvalidInitialize)
26:  if  $\neg \text{verify\_state}((M_1, \dots, M_q), M, A)$  then
27:    return Err(InvalidState)
28:  status  $\leftarrow$  "active"

29: fn verify_state( $[M_1, \dots, M_q], M, A$ )
30:   $[A_1, \dots, A_q] \leftarrow A$ 
31:  for all  $i \in [1, \dots, q]$  do
32:     $M'_i \leftarrow M_i$ 
33:    for all  $\ell \in M_i$  do
34:      for all  $a \in A_i[\ell]$  do
35:         $M'_i[\ell].tail \leftarrow (H(M'_i[\ell].tail), a)$ 
36:         $M'_i[\ell].index \leftarrow M'_i[\ell].index + 1$ 
37:    if  $M'_i \neq M$  then return false
38:  return true

```

**(1) Finalize existing endorsers.** A coordinator interacts with endorsers in  $\mathcal{E}$  to “finalize” their state. In particular, a coordinator invokes a new API supported by an endorser, called **finalize**. It takes as input the new configuration’s public keys  $C_{\mathcal{N}}$  and it outputs  $(M_i, \sigma_i)$ , where  $M_i$  is the endorser’s state and  $\sigma_i$  is a signature on the message  $\langle \text{"finalize"}, id, C_{\mathcal{E}}, C_{\mathcal{N}}, M_i \rangle$  (an endorser signs over  $C_{\mathcal{N}}$  because it is intended to be consumed by endorsers in  $\mathcal{N}$ ). A coordinator waits for a quorum of endorsers in  $\mathcal{E}$  to finalize their state and persists their responses in the untrusted state machine. We refer to the aggregated response as  $R_{exist}$ .

Once an endorser provides a response to finalize, it enters a “finalized” mode where it erases its signing key, and hence it *cannot* process any further requests.<sup>2</sup> It however continues

<sup>2</sup>Nimble’s threat model (Section 4) assumes that active endorsers are not vulnerable (e.g., they do not leak their keys). By erasing a signing key in finalize, Nimble ensures that if an adversary can break the TEE’s guarantees in the future, they cannot recover finalized endorsers’ signing keys.

to respond with its finalized state and the signature on the finalized state, to ensure liveness.

**(2) Initialize new endorsers.** We modify the **initialize** method of the endorser’s state machine described in Section 5.1 to support transferring state (including an existing public identity) to the new endorsers ( $\mathcal{N}$ ). As before, if the coordinator supplies an empty state and an empty prior configuration (i.e.,  $M$  and  $C_{\mathcal{E}}$  are both  $\perp$ ), then this means that this is a new instance of Nimble and the public identity is the hash of the current configuration ( $H(C_{\mathcal{N}})$ ). What is new is that the coordinator could instead supply *any* state ( $M$ ), prior configuration ( $C_{\mathcal{E}}$ ), and public identity ( $i$ ) that it wishes, and the endorsers simply accept that information. The endorsers check that this information is actually safe to use before they start processing requests during the activate function, which we describe next. The initialize function outputs a signature  $\eta_i$  on the message  $\langle \text{"initialize"}, id, C_{\mathcal{E}}, C_{\mathcal{N}}, M \rangle$ .

A coordinator waits for a quorum of endorsers in  $\mathcal{N}$  to initialize their state and persists their responses in the untrusted state machine. We refer to the aggregated response as  $R_{next}$ .

**(3) Activate new endorsers.** We add a new API called **activate** that allows a coordinator to convince an initialized endorser in  $\mathcal{N}$  to start processing requests. The coordinator must provide evidence that it is safe for the endorser to “take over” the initialized identity. An endorser performs a sequence of checks: (1) it checks that a quorum of existing endorsers in  $\mathcal{E}$  have been finalized; (2) it checks that a quorum of new endorsers in  $\mathcal{N}$  have been initialized with the same state; and (3) the state of a quorum of new endorsers is derived from the state of a quorum of existing endorsers by picking ledger tails with the highest positions seen in the quorum.

To prove (1) and (2), the coordinator provides  $R_{exist}$  and  $R_{next}$  respectively. To prove (3), the coordinator provides additional blocks ( $A$ ) that can be appended to the tail of each of the ledgers that were supplied by existing endorsers when they called finalize ( $M_i$ ) such that the resulting state equals  $M$ . This check is in the **verify\_state** function. An honest coordinator can find these blocks in the untrusted state machine.

**Verifying receipts in the presence of reconfigurations.** Suppose that a client goes offline and Nimble executes several reconfigurations. We now discuss how such a client can verify receipts produced by Nimble. Recall that a client retains the public identity of Nimble ( $id$ ).

Observe that Nimble’s reconfiguration protocol ensures that endorsers in a Nimble instance *always* use the same public identity  $id$ . Furthermore, an endorser’s signature covers, in addition to its response, both the public identity  $id$  and the public keys of endorsers in its own configuration (i.e.,  $C_{curr}$ ). We extend the coordinator’s APIs so a client can use them to retrieve  $C_{curr}$  and each endorser’s attestation report. This allows a client to (lazily) learn the public keys of endorsers in the latest configuration as well as verify that those public keys belong to legitimate endorsers. Finally, a client does the following checks to verify a receipt: (1) public keys in the



| Component   | Trusted? | Language | SLoC |
|-------------|----------|----------|------|
| Coordinator | No       | Rust     | 3564 |
| Endorser    | Yes      | Rust     | 1843 |
| Endorser    | Yes      | C++      | 559  |
| Endpoint    | Yes      | Rust     | 517  |

FIGURE 2—Implementation of Nimble (SLoC) (excluding existing libraries and crates used by Nimble)

receipt are in  $C_{curr}$ ; (2) signatures are valid when verified with the known  $id$  and  $C_{curr}$  (as well as other information specific to a request); (3) there is a quorum of valid signatures based on the number of public keys in  $C_{curr}$ .

**Lemma 5.3.** *Assuming the integrity and confidentiality guarantees provided by TEEs for executing the specified endorser state machine and standard cryptographic hardness assumptions, at any point in time, if Nimble produces a valid receipt for an append operation  $O$  using endorsers in  $\mathcal{E}$ , and if a coordinator activates a majority of endorsers in  $\mathcal{N}$  with state  $M$ , then  $M$  must contain the effects of executing  $O$ .*

*Proof (sketch).* Consider an append request  $O$  for which Nimble produces a valid receipt using endorsers in  $\mathcal{E}$ . This means that a majority of endorsers in  $\mathcal{E}$  applied  $O$  and provided a signature on the same response (let  $Q$  denote that majority). Furthermore, any majority of endorsers in  $\mathcal{E}$  must contain at least one endorser from the set  $Q$ .

Now, by the premise, the coordinator successfully activates a majority of endorsers in  $\mathcal{N}$ . This means that the coordinator must have called their initialize and activate methods such that all checks in the activate method pass. By assumptions about TEEs and cryptography, this means that a majority of endorsers in  $\mathcal{N}$  must have been given with  $(R_{exist}, A, R_{next})$  such that all checks in the activate method pass. Again, by the aforementioned assumptions, the only feasible way to achieve this is by calling finalize on a majority of endorsers in  $\mathcal{E}$  to obtain a valid  $R_{exist}$ . From the aforementioned reasoning, at least one of states retrieved from a majority of endorsers in  $\mathcal{E}$  must contain the effects of applying  $O$ . Thus,  $M$  provided to a majority of endorsers in  $\mathcal{N}$  contains the effects of applying  $O$ . Finally, by design, once an endorser in  $\mathcal{E}$  returns a response to the finalize method, it *cannot* process any request, as a result, once  $R_{exist}$  is generated, no valid receipt can be generated by using a majority of endorsers in  $\mathcal{E}$ .  $\square$

**Lemma 5.4.** *When the service is honest and during sufficiently long periods of synchrony, if a majority of endorsers in  $\mathcal{E}$  and  $\mathcal{N}$  are live, then a coordinator that can be subject to crash failures can eventually obtain a majority of endorsers in  $\mathcal{N}$  to start processing requests.*

*Proof (sketch).* We first argue that the claim holds when a coordinator does not experience crashes. We then argue that a coordinator that restarts can still complete the protocol by using state persisted in the untrusted state machine.

We need to establish that the coordinator *can* provide inputs that pass checks in the activate method of the endorser state machine. By inspection, if the coordinator follows its prescribed protocol, one can see that nearly all of the checks in the invoked activate method pass on an endorser state machine that holds a signing key where the corresponding public key is in the sequence  $C_{curr}$ . We now argue that `verify_state` check passes too. Suppose that the coordinator computes  $M$  as specified in the protocol. When the service is honest, for every append request processed by the service, it is first applied on the untrusted state machine (which is linearizable and crash fault-tolerant) and then applied on each endorser in the same order (endorser’s state machine is also linearizable). However, for any given ledger, some endorsers may be “lagging behind” others since Nimble requires only a majority of endorsers to process an append. This implies that an honest coordinator can retrieve blocks from the untrusted state machine, and construct  $A$  such that `verify_state` $((M_1, \dots, M_q), M, A) = \text{true}$ .

Now, consider the case where a coordinator may crash. Observe that each step persists state in the untrusted state machine. When a coordinator restarts, it can examine the state in the untrusted state machine to identify the step in which it failed and repeatedly retry steps in the specified reconfiguration protocol. Furthermore, by design, all APIs of an endorser (e.g., initialize, finalize) are idempotent. Even if a coordinator finished a step but failed before logging state into the untrusted state machine, a new coordinator can safely retry the step. As a result, a coordinator that repeatedly retries eventually activates a quorum of endorsers in  $\mathcal{N}$ , which then can produce valid receipts for clients’ requests.  $\square$

## 6 Implementation

Nimble is available as an open-source project [3]. Nimble’s implementation is in Rust. In addition to the Rust-based endorser, we implement an endorser in C++ using the Open Enclave SDK [4]. The Rust endorser runs inside a confidential VM supported by AMD SEV-SNP, and the OpenEnclave-based endorser runs inside Intel SGX. Both a coordinator and an endorser run as microservices, each exposing an RPC interface. To ease the adoption of Nimble, we implement an *endpoint* that exposes a REST API. The endpoint implements the client-side verification logic (Section 5.2) and runs inside a confidential VM (i.e., a client essentially outsources all of its verification tasks to the endpoint). With an endpoint, a client only needs to perform remote attestation to ensure that the right code runs, and establish a secure channel with it.

For cryptographic primitives, endorsers use SHA-256 for hash functions and ECDSA with P-256 for signatures, both implemented by OpenSSL.

Figure 2 shows the lines of code for each component in Nimble. The Rust-based endorser implements the full protocol described in Section 5, while the C++ implements only the core protocol (no reconfiguration).

We implement several optimizations. First, a coordinator waits only for a quorum of endorsers to provide a matching

response (this helps reduce latency, especially when a minority of endorsers is deployed in a remote region, is slow, or is disconnected). Second, an endorser stores a copy of the tail node (rather than its hash), so for read operations, this allows a coordinator to avoid a round trip with the storage service.

Nimble’s implementation supports reconfigurations to replace failed endorsers. However, the coordinator microservice does not proactively invoke the reconfiguration protocol. Instead, it exposes additional *control* APIs that allow a monitoring process to trigger the addition or removal of endorsers. In a full deployment of Nimble, we expect to make use of a monitoring infrastructure to invoke these control APIs.

## 7 Evaluation

This section answers the following evaluation questions:

- What is the latency and throughput of Nimble operations, and how do they depend on the underlying storage or TEE technology used by Nimble?
- How does Nimble’s TCB compare to alternative solutions?
- What is the cost of a reconfiguration, and how does it scale with the number of ledgers in Nimble?
- How difficult is it to port a real application to use Nimble and what overheads does Nimble introduce?

### 7.1 Experimental setup

We run all of our experiments on Azure. We run endorsers on three different machines, each on a different availability zone to ensure that a server, rack, or even an entire data center failure does not cause a loss of a quorum of endorsers. When we run endorsers in Intel SGX, we use Azure DC32s v3 instances; when we run endorsers in AMD SEV-SNP, we use Azure DC32as v5 instances. The coordinator runs in Azure D48ads v5 instances, as does our client (a workload generator). Finally, we deploy several endpoints that run inside AMD SEV-SNP and execute verification logic. We use an Azure load balancer to route requests among the endpoints, and direct all client requests to this load balancer.

Given that Nimble inherits the performance of its underlying storage, we evaluate two configurations:

- *An in-memory key-value store*: An unreplicated in-memory key-value store that does not tolerate failures. This key-value store has low access latency and high throughput. It serves as a best-case scenario for Nimble. In a real deployment of Nimble, this store could be replaced with existing replicated in-memory key-value stores that provide high throughput and low latency [23, 43, 44].
- *Azure table with geo-replication*: Azure storage with the strongest replication guarantees enabled (RA-GZRS). It has lower throughput and higher latency than our in-memory key-value store, and suffers from high tail latency since it is a multi-tenant cloud service with no SLAs.

### 7.2 Latency and throughput of Nimble

We start by conducting a series of microbenchmarks on Nimble. To generate workloads, we use *wrk2* [50], a popular constant-load open-loop workload generator. The resulting workload is sent to our Azure load balancer which is then split across our REST endpoints. We measure the median and 90-th percentile latencies, as well as the throughput achieved by Nimble on its different configurations.

Figure 3 depicts the results. Figure 3a shows the performance of Nimble when using AMD SEV-SNP endorsers and our in-memory key-value store. In this configuration, the median latency of all operations is under 2.5ms, and the 90-th percentile latency is under 3.2 ms. This latency is possible due to the fast communication between machines inside Azure data centers, even if the endorsers and coordinator are in different availability zones. Append and read throughput both peak at around 50K req/sec, which is quite significant given that Nimble’s endorsers process and sign *individual* requests; we do not do any batching in this experiment. The bottleneck is indeed computational and comes from the cryptographic operations performed by the endorsers.

Figure 3b shows the performance of Nimble when using AMD SEV-SNP endorsers and Azure table storage. In this configuration we observe two things: (1) the higher storage latency plays a key role for append operations, leading to median latencies of around 30–40 ms. More significantly, tail latencies are very high (sometimes up to 2 seconds), owing to the fact that we use a shared service without guaranteed SLAs. The append throughput performance is significantly worse than our in-memory counterpart, reaching around 2,600 reqs/sec. Here the bottleneck is no longer computational and is instead Azure storage which has an account-wide throughput limit of 20K entities/sec; in Nimble, every append accesses multiple rows (entities) in Azure Table to provide the required untrusted state machine semantics (Section 5.1), which hits this limit. Reads are not impacted by these salient properties of storage because of our fast-reads optimization (Section 6).

Figure 3c shows the performance of Nimble when using Intel SGX endorsers and our in-memory key-value store. The performance is lower than AMD SEV-SNP endorsers because our SGX endorsers must continuously cross between the untrusted host that runs the networking stack and the enclave, in addition to the hardware being completely different. Finally, we omit the SGX endorsers and Azure Table configuration because the performance is very similar to that of Figure 3b, owing to Azure storage being the bottleneck.

### 7.3 Comparison of TCB size

A key component of Nimble is its relative simplicity. We therefore ask how Nimble compares to similar proposals. Figure 2 gives the breakdown of the complexity of Nimble’s different components (we use TCB as a proxy for it), and Figure 4 compares it to other works. The key take away is that the

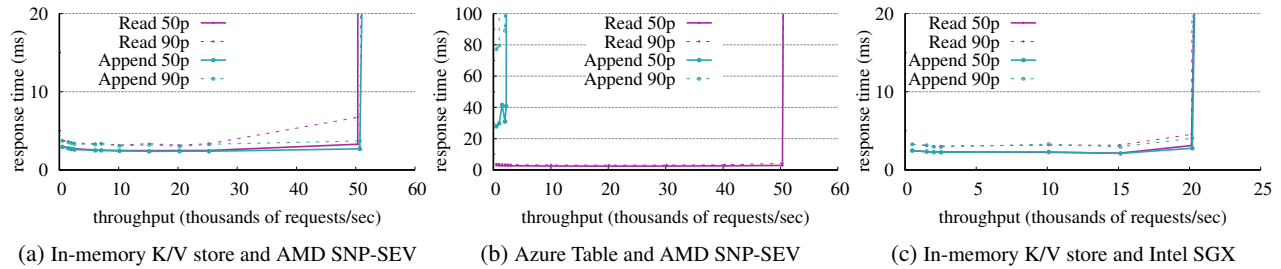


FIGURE 3—Microbenchmark of the different operations supported by Nimble. Time is measured end-to-end from the perspective of the client and includes the time needed for the endpoint to verify the signatures provided by Nimble on behalf of the client.

| System             | TCB   | restarts?        | reconfig? |
|--------------------|-------|------------------|-----------|
| ROTE [35]          | 1.1K  | No               | No        |
| Narrator [39]      | 5.1K  | Yes              | No        |
| TEEMS [22]         | 11.0K | Yes              | No        |
| CCF [41]           | 55.5K | Yes <sup>†</sup> | Yes       |
| Nimble (this work) | 2.3K  | Yes <sup>†</sup> | Yes       |

FIGURE 4—Source lines of code (SLoC) comparison of Nimble to other works that provide a fault-tolerant rollback detection service, and whether they can support a replica that restarts (fails and then comes back), as well as replacing the set of replicas. For ROTE, Narrator, and TEEMS we use the numbers from the papers. For CCF we use numbers provided by the authors. Note that this table should be treated *qualitatively* since these systems are implemented in different languages with different coding styles, libraries, etc. The takeaway is that ROTE, Narrator, and Nimble are “simple”; TEEMS is moderately complex as it implements an entire replication protocol within the TCB; and CCF is more complex since it implements a replication protocol in addition to logic that handles blockchain smart contracts. <sup>†</sup>CCF and Nimble can handle replica restarts by treating them as new replicas and engaging reconfiguration.

| # of ledgers | median reconf. time | total communication |
|--------------|---------------------|---------------------|
| 100K         | 805 ms              | 175.59 MB           |
| 200K         | 1.53 sec            | 337.65 MB           |
| 500K         | 3.72 sec            | 881.62 MB           |
| 1M           | 7.14 sec            | 1.68 GB             |

FIGURE 5—Total reconfiguration time (median across 10 runs) and the total amount of communication between the coordinator and the old and new endorsers (measured with `tcpdump`).

complexity of Nimble’s TCB is similar to that of ROTE [35] and Narrator [39], despite the fact that Nimble supports reconfiguration and these prior systems do not. When compared to TEEMS [22] or CCF [41], Nimble is significantly simpler. Indeed, it is precisely this simplicity that allowed us to formally prove the safety of the core protocol of Nimble using the Ironfleet methodology [27] and the Dafny program verifier [31]. Doing the same for these other works that include an entire consensus protocol in their TCB is a daunting task.

#### 7.4 Cost of reconfiguration

One of the key innovations in Nimble is the ability to securely reconfigure from one set of endorsers to another. As

we explain in Section 5.2, this process requires “finalizing” existing endorsers, which effectively stops request processing while the reconfiguration takes place. Hence, reconfiguration time impacts the availability of Nimble. There are two factors that affect this time: (1) the number of ledgers in the system, and (2) the difference between the number of ledger entries processed by the endorsers. We find that (1) is the dominating factor given that the fast network and endorsers running on similar hardware in our experimental setup lead to small differences in which ledger entries they have processed.

To measure the impact of (1), we conduct an experiment where we populate the system with a varying number of ledgers and then induce a reconfiguration to replace an existing set of three endorsers to a brand new set of three endorsers running on different machines (also on three different availability zones). Figure 5 depicts the results for both total time and network communication. We observe a near-linear cost increase in reconfiguration time in terms of the number of ledgers. This cost comes from a variety of factors: (i) hashing of the state at existing endorsers; (ii) determining which state to initialize the new endorsers and transferring that state; and (iii) hashing and verification of the provided state at the new endorsers (e.g., the `verify_state` method).

**Balancing costs.** Given that Nimble’s cost of reconfiguration is high when the system supports many ledgers, and such cost translates directly to service unavailability, one possibility is to partition the ledger space so that disjoint sets of endorsers are responsible for different sets of ledgers. In this manner, if an endorser in one of the partitions fails, the provider can perform a reconfiguration that changes only the ledgers within this partition—without needing to touch the other partitions. Of course, disaster scenarios such as an entire availability zone going down could still require endorsers in all partitions to be swapped, but this is a rarer event.

#### 7.5 Integrating applications with Nimble

One important consideration with a system like Nimble is how would existing applications use it. To answer this question, modify the Hadoop Distributed File System (HDFS). We choose HDFS because (1) it has a lot of state at many different components and (2) any cloud customer who runs a data analytics application that uses HDFS is vulnerable to rollback

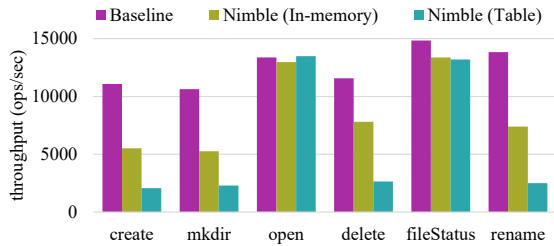


FIGURE 6—Results of NNThroughputBenchmark on an HDFS deployment (Baseline) and a deployment of Nimble-HDFS.

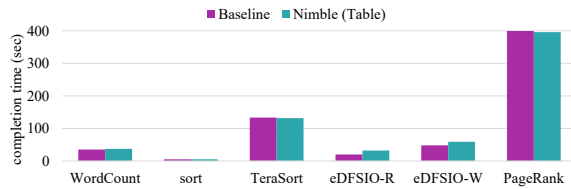


FIGURE 7—Results of Intel’s HiBench with a dataset scale set to “large” on a MapReduce deployment running on top of standard HDFS (Baseline) and a Nimble-HDFS backed by Azure Table.

attacks today—even if HDFS and their application runs on confidential computing servers. Hence, a rollback-resistant HDFS would provide significant benefit.

We spent three person months modifying HDFS to integrate with Nimble, for a total of 1,689 lines of Java. We discuss the specifics in [11, Appendix B]. At a high level, we observe that HDFS logs data and metadata in order to recover from crash failures. These logs are committed to disk either synchronously or periodically in batches (in which case the system might lose the latest uncommitted state during a failure). While these logs are in memory, we can protect them by running HDFS’s namenode and datanode inside TEEs. However, as soon as they are written to disk, they are vulnerable to rollback attacks. We identified all such events in HDFS and replaced them with the approach in Section 3.1. The result is Nimble-HDFS, a version of HDFS that detects rollbacks.

To measure this overhead, we provision two Azure F64s v2 machines, one to run the namenode and the other the datanode of HDFS or Nimble-HDFS. We then run Hadoop’s NNThroughputBenchmark [9], which is a standard benchmark that measures the performance of HDFS operations such as `create`, `mkdir`, etc. We configure Nimble-HDFS to append an entry to Nimble every 100 operations. At the peak throughput, the window of vulnerability is tens of ms.

Figure 6 depicts our results. For some operations, Nimble-HDFS has no overhead over the baseline, particularly those that do not append entries to Nimble (deviations are basically experimental noise). For others, Nimble-HDFS introduces up to a 2 or 3× overhead over the baseline, depending on the backing store. This cost comes from computing digests, sending them over HTTP to Nimble’s endpoint, and flushing operations to disk before moving on.

At first glance, these added costs might appear problematic.

But the reality is that the overhead of Nimble-HDFS is minimal for real applications. To demonstrate this, we run Intel’s HiBench Suite [2], which consists of big data applications that run on top of MapReduce. We configure MapReduce to use either standard HDFS or Nimble-HDFS. The results are in Figure 7. As we can see, there is essentially no difference in the job completion time for most of these applications when using Nimble; the exception is the extended DFSIO benchmark which is I/O heavy and is meant to measure the performance of the underlying HDFS instance.

## 8 Related work

This section discusses works that directly relate to Nimble; while there are many other works on building untrusted storage systems [14, 16, 24, 33, 34, 36], our focus here is on projects that guarantee linearizability.

**Rollback protection.** Many TEEs (e.g., Intel SGX) support *sealing*. Sealing enables applications running inside TEEs to encrypt and sign their state with secret keys known only to the TEE, prior to storing in untrusted disk. Sealing alone does not provide rollback protection, but one can additionally use monotonic counters supported by TEEs. There are several drawbacks to this combination. First, operations on counters are slow (e.g., increment latencies are 80–250ms) and wear out in a few days [12, 35], though recent systems like SPEICHER [12] partially address this issue. Second, monotonic counters are not as secure as expected (e.g., removing the BIOS battery or reinstalling TEE software often resets these counters). Finally, monotonic counters are specific to a given machine so a crashed application cannot be launched on a different machine—which is unacceptable in cloud settings.

Memoir [40] provides rollback protection by maintaining a history of application requests in an append-only hash chain (which itself is in an untrusted storage) and tracking tail of the chain in a trusted non-volatile memory supported by a TPM. If an application restarts, it uses state in the trusted non-volatile memory and the hash chain to reconstruct its state. Ariadne [45] similarly uses a TPM’s non-volatile memory but with a different abstraction (counter instead of hash chain). The challenge with these approaches in cloud settings is that, if the TPM or its machine fails, the system becomes unavailable. Nimble solves this challenge by developing a fault-tolerant version of Memoir that stores the state in several TEEs’ volatile memory and supports reconfiguration.

ROTE [35], Narrator [39], and TEEMS [22] are similar to Nimble in that they propose a solution to help confidential applications in TEEs detect rollbacks. The main difference with Nimble is that these works lack a reconfiguration protocol, so there is no obvious way to add or remove replicas.

CCF [41] provides rollback-resistance and supports reconfiguration but it is significantly more complex than Nimble and has a very large TCB since it is designed to run and validate smart contracts and other blockchain constructs.



Kaptchuk et al. [29] formalizes the interactions of a TEE with an append-only ledger as a way to provide rollback protection. A key distinction with Nimble is that their work assumes the existence of the ledger, whereas Nimble focuses on building the ledger itself.

Wang et al. [49] study pitfalls with running a crash fault-tolerant replication protocol inside TEEs to achieve a Byzantine fault-tolerance. In particular, they describe concrete attacks including rollback attacks on state kept by individual nodes on their local disks. For rollback attacks, they propose a solution based on ROTE [35] that inherits its drawbacks.

**Replicated systems with a small TCB.** A2M [18] and Trinc [32] propose trusted primitives for nodes in a distributed system to prevent malicious nodes from equivocating (i.e., sending conflicting messages to different nodes). Unfortunately, these trusted primitives do not aim to provide fault-tolerance on their own. A straightforward use of a replication protocol to add fault-tolerance (including reconfigurations) results in a large TCB, analogous to CCF's.

A recent line of work focuses on using minimal trusted primitives to improve various aspects of replication protocols. Yandamuri et al. [52] use a Trinc-type minimal trusted hardware in communication-efficient Byzantine fault-tolerant protocols [10, 53] to preserve communication efficiency while achieving improved fault thresholds. Similarly, Damysus [21] separates safety and liveness concerns in HotStuff [53] and describes minimal trusted components that improve fault thresholds. Hybster [13] and FlexiTrust [26] observe that Trinc-type trusted hardware forces sequential invocations of consensus instances, so they introduce variants that support parallel instances and achieve better performance. Unfortunately, these works do not yet support reconfiguration.

## 9 Discussion

### 9.1 Disaster recovery

Recall that Nimble runs a set of endorsers inside TEEs and receipts consist of signatures from a quorum of endorsers. This raises a question: what happens if Nimble loses a majority of endorsers? If the endorsers are alive but disconnected (e.g., network partition), then Nimble will experience unavailability until a quorum is accessible again. If the endorsers actually crashed and their volatile state is gone, then we refer to this scenario as a *total disaster*. This could happen for a number of reasons. Perhaps the service provider experiences a massive attack or a natural disaster takes down multiple datacenters.

The good news is that total disasters do not affect safety properties like freshness. By design and implementation, endorsers cannot be restarted. If Nimble loses a majority of its endorsers, then there is no longer a quorum of endorsers that can sign responses that a client will accept. The bad news is that this leads Nimble to lose liveness.

There are some ways to reduce the chance of total disasters. The most important one is with a reconfiguration protocol

so that failures do not pile up and cause the system to lose a quorum of endorsers. This is why we developed one for Nimble. Second, deploy endorsers in different fault domains (cloud providers already do this for their replicated systems).

Even with such measures, total disasters could still occur. Unfortunately, there is no “playbook” for how to proceed. An option is for customers to periodically snapshot the tails of their ledgers and store them in some location they trust. After a total disaster, the customer can ask the service provider if they have a snapshot that is more recent than the one they have (the customer can check that it includes more updates than their own and in fact the snapshot is legitimate by verifying receipts). Then, customers can explicitly ask the service to create a new instance of Nimble that starts with that agreed-upon snapshot. Of course, if the snapshot is stale then the system will not reflect the most recent operations (i.e., the responses will not be fresh), but observe that a provider cannot do this unilaterally: the customer must explicitly ask for it. If a customer does not want to maintain snapshots, then an option is to accept a snapshot provided by the provider (the customer can still verify that some prior endorsers signed those tails). This might be acceptable in extreme situations such as when the total disaster was due to a public natural disaster that took down datacenters where endorsers were deployed.

### 9.2 TCB changes

Our description has so far assumed that endorsers' trusted code does *not* change (i.e., when verifying receipts  $R_{exist}$  and  $R_{next}$ , an endorser checks that its own measurement matches the measurements of an existing quorum of endorsers and those of a new quorum of endorsers). But what if that trusted code needs to be updated? For example, if there was an update to a library or the attestation verification procedure changed.

To address this, we sketch a solution, which omits the aforementioned check requiring measurements to match; instead customers have to do certain checks. Specifically, the service provider persists  $R_{exist}$  and  $R_{next}$  whenever a reconfiguration occurs, along with a copy of the code running in an endorser (and other configuration information to reproduce binaries loaded inside TEEs). The provider then uses this information to prove to customers that all configuration changes (including code changes in the TCB) were legitimate. Customers must audit code changes and verify attestation reports, and decide whether to accept the latest set of endorsers.

Note that the above proposal crucially assumes that whenever the provider reconfigures from an existing set of endorsers  $\mathcal{E}$  to a new set of endorsers  $\mathcal{N}$  with a *new* trusted code, a quorum of endorsers in  $\mathcal{E}$  was not exploited by an adversary *before* the reconfiguration. This is because after a quorum of endorsers in  $\mathcal{E}$  finalize their state (which is necessary for reconfiguration), they erase their signing keys (Footnote 2). However, there is no known way to prove that the trusted code was updated before it was exploited by an adversary.

## Acknowledgments

We thank Leslie Lamport, Melissa Chase, the OSDI reviewers, and our shepherd, Brad Karp, for their thorough comments and helpful conversations. We thank Jonathan Lee and Jay Lorch for helpful discussions when the project began, and Amaury Chamayou and Cédric Fournet for helping us better understand CCF. We also thank Ahmad Abdullateef, David Altobelli, Anil Bazaz, Pushkar Chitnis, Greg Kostal, Hervey Wilson, and Sergio Wong for helping us identify requirements that Nimble must support. Basu and Jaeger were supported in part by the U.S. Army Combat Capabilities Development Command Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and NSF grant CNS-1816282. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

## References

- [1] Big data analytics on confidential computing with Apache Spark on Kubernetes. <https://learn.microsoft.com/en-us/azure/architecture/example-scenario/confidential/data-analytics-containers-spark-kubernetes-azure-sql>.
- [2] HiBench Suite: The bigdata micro benchmark suite. <https://github.com/Intel-bigdata/HiBench>.
- [3] Nimble: Rollback Protection for Confidential Cloud Services. <https://github.com/Microsoft/Nimble>.
- [4] Open Enclave SDK. <https://github.com/openenclave/openenclave>.
- [5] Reference architecture for privacy preserving machine learning with Intel SGX and TensorFlow serving. <https://www.intel.com/content/www/us/en/developer/articles/technical/privacy-preserving-ml-with-sgx-and-tensorflow.html>.
- [6] Technology preview for secure value recovery. <https://signal.org/blog/secure-value-recovery/>.
- [7] PySyft, PyTorch and Intel SGX: Secure aggregation on trusted execution environments. <https://blog.openmined.org/pysyft-pytorch-intel-sgx/>, 2020.
- [8] Fledge services for chrome and android. <https://developer.chrome.com/blog/fledge-service-overview/>, 2022.
- [9] Hadoop benchmarking. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/Benchmarking.html>, 2022.
- [10] I. Abraham, D. Malkhi, and A. Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. arXiv, 2018.
- [11] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. Setty, and S. Singanamalla. Nimble: Rollback protection for confidential cloud services (extended version). Cryptology ePrint Archive, Paper 2023/761, 2023.
- [12] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: securing LSM-based key-value stores using shielded execution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.
- [13] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.
- [14] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2017.
- [15] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.
- [16] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.
- [17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [18] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiawicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, page 189–204, 2007.
- [19] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016. <https://eprint.iacr.org/2016/086>.
- [20] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the USENIX Security Symposium*.
- [21] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. DAMYSUS: streamlined BFT consensus leveraging trusted components. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.
- [22] B. Dinis, P. Druschel, and R. Rodrigues. Rr: A fault model for efficient tee replication. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.
- [23] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.
- [24] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3), 1996.
- [26] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi. Dissecting BFT consensus: In trusted components we trust! In

*Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

- [27] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.
- [28] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.
- [29] G. Kaptchuk, I. Miers, and M. Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.
- [30] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.
- [31] R. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.
- [32] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
- [33] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
- [34] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [35] S. Matetic, M. Ahmed, K. Kostiaainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *Proceedings of the USENIX Security Symposium*, 2017.
- [36] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.
- [37] moxie0. Technology preview: Private contact discovery for Signal. <https://signal.org/blog/private-contact-discovery/>, 2017.
- [38] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [39] J. Niu, W. Peng, X. Zhang, and Y. Zhang. Narrator: Secure and practical state continuity for trusted execution in the cloud. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
- [40] B. Parno, J. Lorch, J. J. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011.
- [41] M. Russinovich, E. Ashton, C. Avanesians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna, J. Maffre, T. Moscibroda, K. Nayak, O. Ohrimenko, F. Schuster, R. Schwartz, A. Shamis, O. Vrousou, and C. M. Wintersteiger. CCF: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.
- [42] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [43] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2019.
- [44] W. Shen, A. Khanna, S. Angel, S. Sen, and S. Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.
- [45] R. Stackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *Proceedings of the USENIX Security Symposium*, 2016.
- [46] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.
- [47] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [48] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. Albassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom. SoK: SGX.Fail: How stuff get eXposed. <https://sgx.fail>, 2022.
- [49] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
- [50] wrk2: A constant throughput, correct latency recording variant of wrk. <https://github.com/giltene/wrk2>.
- [51] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [52] S. Yandamuri, I. Abraham, K. Nayak, and M. K. Reiter. Communication-efficient BFT protocols using small trusted hardware to tolerate minority corruption. Cryptology ePrint Archive, Paper 2021/184, 2021.
- [53] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.
- [54] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

# *Kerveros*: Efficient and Scalable Cloud Admission Control

Sultan Mahmud Sajal<sup>1,3</sup> Luke Marshall<sup>1</sup> Beibin Li<sup>1</sup> Shandan Zhou<sup>2</sup> Abhisek Pan<sup>2</sup>  
Konstantina Mellou<sup>1</sup> Deepak Narayanan<sup>1</sup> Timothy Zhu<sup>3</sup> David Dion<sup>2</sup>  
Thomas Moscibroda<sup>2</sup> Ishai Menache<sup>1</sup>

<sup>1</sup>Microsoft Research   <sup>2</sup>Microsoft Azure   <sup>3</sup>Pennsylvania State University

## Abstract

The infinite capacity of cloud computing is an illusion: in reality, cloud providers cannot always have enough capacity of the right type, in the right place, at the right time to meet all demand. Consequently, cloud providers need to implement admission-control policies to ensure accepted capacity requests experience high availability. However, admission control in the public cloud is hard due to dynamic changes in both supply and demand: hardware might become unavailable, and actual VM consumption could vary for a variety of reasons including tenant scale-outs and fulfillment of VM *reservations* made by customers ahead of time. In this paper, we design and implement *Kerveros*, a flexible admission-control system that has three desired properties: i) high computational scalability to handle a large inventory, ii) accurate capacity provisioning for high VM availability, and iii) good packing efficiency to optimize resource usage. To achieve this, *Kerveros* uses novel bookkeeping techniques to quickly estimate the capacity available for incoming VM requests. Our system has been deployed in Microsoft Azure. Results from both simulations and production confirm that *Kerveros* achieves more than four nines of availability while sustaining request processing latencies of a few milliseconds.

## 1 Introduction

Cloud capacity appears to be limitless. However, in reality, cloud providers need to deal with the limitations of datacenters with a finite number of machines while respecting contractual service-level agreements (SLAs) that provide availability guarantees to customer VMs. These SLAs might be severely compromised if the provider runs out of resources. Additionally, users expect predictability: not being able to launch new VMs when required can critically impact a customer’s business [31]. An admission-control system is thus necessary to ensure cloud providers do not *overcommit* resources, provide seamless elasticity, and are robust to capacity loss due to failures. This paper describes the design and implementation of *Kerveros*: a scalable and efficient

admission-control system for Microsoft Azure.

Admission control in the cloud is hard because it needs to account for a variety of fluctuations in both supply and demand across a large number of workload and machine types. On the supply side, datacenter machines and racks regularly fail at scale and maintenance tasks like software updates may also require rebooting machines. The cloud provider needs to maintain high availability amidst both deterministic and stochastic events by keeping enough resources free to provide seamless failover if necessary. On the demand side, to facilitate predictability, cloud providers have recently introduced the notion of “reserved resources” (or capacity *reservations*) [4, 6, 7, 20, 35] to guarantee the availability of capacity in the future. With such reservations, customers pay to have the provider set aside resources that can be claimed later. Accordingly, admission-control systems need to accommodate both on-demand VM requests and reservations.

The goal of our cloud admission-control system is to ensure that it simultaneously achieves *high computational scalability* and *a low rate of SLA violations* while avoiding inefficient capacity usage (which can lead to negative margins). One possible approach to accomplish this is to re-use existing VM allocators [21, 43] to directly allocate space for reservations, maintenance, and potential tenant growth (scale-outs). However, such a “placeholder” approach is slow and inherently inflexible, since the capacity allocated to such placeholders cannot be immediately used for other incoming VM requests that can pack well in the reserved space.

Instead, our solution is based on an approximate *buffer* approach that avoids early binding of placeholders. At a high level, we maintain buffers of resources for handling failures, maintenance, growth, reservations, etc., and track the total number of remaining resources after accounting for currently running VMs *and* buffers. Implementing this idea involves several algorithmic challenges, such as reasoning about the capacity required by VMs with multi-dimensional resource demands (e.g., CPU, memory, disk), quantifying the impact of buffers on a large variety of possible VM requests (e.g., Azure has more than 1000 VM types; see Figure 3 for details),



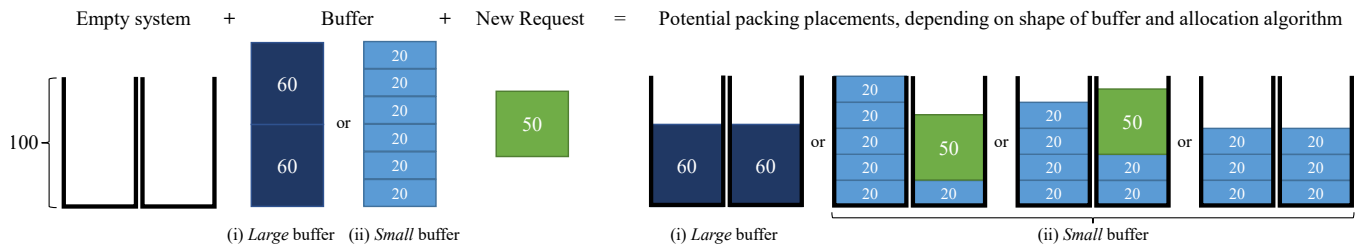


Figure 1: Consider an empty cluster of two machines with capacity 100 units each. Suppose one reservation with total size of 120 has already been accepted. We consider two cases: (i) the reservation consists of two VMs of size 60; (ii) the reservation consists of six VMs of size 20. A new single VM request of size 50 enters the system. Note that by solely taking the available capacity units into account (i.e.,  $200 - 120 = 80 \geq 50$ ), the system would accept the new request in both cases. However, when accounting for packing considerations, the request should be accepted only in case (ii). Further, if the underlying allocation algorithm was designed to load balance the VMs corresponding to a reservation, then the request should be rejected in both cases.

simultaneously accounting for buffers defined at different levels of the datacenter hierarchy (e.g., cluster vs. zone), and mimicking how the allocator would place VMs belonging to admitted reservations. Indeed, simple back-of-the-envelope buffer calculations might result in two undesired scenarios: (i) rejecting a request that can actually fit in the cloud, or worse, (ii) accepting a request at the cost of compromising reservation and availability guarantees; Figure 1 illustrates this scenario.

To address these challenges, we introduce a novel admission control mechanism which relies on *Allocable VM (AV)* counts, a bookkeeping technique for quickly determining the capacity available for an incoming VM request or reservation. The AV count, defined per VM type, quantifies the number of VMs that can fit in the inventory at a given time, and reduces the multi-dimensional problem into a formulation with a single dimension (the AV count). *Kerveros* exposes AV counts to the allocator, which can then accept or reject incoming VM or reservation requests based on a simple comparison (i.e., is the requested capacity less than the current AV count?). Importantly, this allows the allocator to respond to requests with high throughput and low latency, a critical requirement for extreme-scale VM allocation platforms [21, 43].

To enable these highly efficient capacity checks, we design the Conversion Ratio Algorithm (CRA), which allows *Kerveros* to quickly translate all buffer sizes to a common unit: the AV count of the incoming request’s VM type. We supplement CRA with a data-driven Linear Adjustment Algorithm (LAA), which periodically emulates the allocator to reduce potential biases of CRA. We implement these algorithms in Azure’s resource-allocation platform, while selectively reusing and enhancing existing allocator infrastructure (e.g., data stores, request handling agents, etc.). We further accelerate the algorithm with a caching layer that allows for incremental AV count updates.

Our results from both simulation and production measurements demonstrate that *Kerveros* sustains at least four nines of availability without any significant degradation

in request-processing throughput. Our admission-control strategy estimates the available capacity with less than 1% error at the 95th percentile. Importantly, this level of accuracy enables Azure to avoid capacity wastage, leading to high return on investment (ROI). We emphasize that for today’s global-scale cloud providers, even a 1% improvement in such a capacity-efficiency metric can be worth 100s of millions of dollars in saved hardware expenditure, translating to sizable impact on the cloud provider’s bottom-line margin.

To the best of our knowledge, this paper is the first to describe the design, implementation and evaluation of an admission-control system deployed in a large public cloud. Prior research on datacenter resource management (e.g., Protean [21] and Borg [43, 45]) focuses mostly on on-demand VM placement. The closest work available is Meta’s RAS system [34], which partitions resources at the granularity of machines to different sub-organizations and periodically re-assigns machines across partitions by solving a mixed integer linear program. While this approach can suit a first-party workload with a modest number of partitions, it is less efficient for dynamic public-cloud workloads (§5).

In summary, our main contributions are:

- The design of scalable and efficient admission-control algorithms for a large and heterogeneous public cloud inventory (§3).
- A robust system design that separates the admission-control logic from the components that enforce it, allowing for latencies of a few milliseconds for admission and placement decisions (§4). Our system has been successfully deployed in Microsoft Azure.
- Our extensive evaluation using measurements from both simulations and production demonstrate that our design achieves scalable and accurate admission control (§5).
- Supplementary to this paper, we release a new trace that can be used by the research community to design and test different packing and admission-control algorithms: <https://github.com/Azure/AzurePublicDataset>.

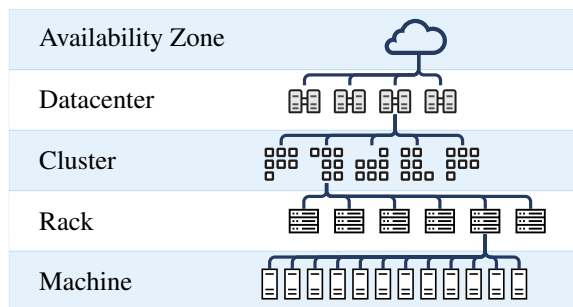


Figure 2: Cloud topology. A zone contains one or more datacenters that house a heterogeneous mix of clusters. Each cluster is comprised of homogeneous machines organized in racks. Racks provide isolation for fault tolerance.

## 2 Background and Motivation

Microsoft Azure is a large public geo-distributed cloud provider with a massive global footprint. Resources in Azure are organized into regions, each of which consists of one or more availability zones (Figure 2). Every zone contains one or more datacenters. Each datacenter is further divided into clusters and racks. Each cluster has a homogeneous set of machines (or servers); however, a zone can have a heterogeneous mix of clusters. As Figure 3 shows, zones can have tens of different hardware types. A zone can have hundreds of thousands of machines, while a cluster is much smaller (at most a few thousand machines).

Each zone has its own allocation service (or simply, *allocator*) that assigns VMs to physical machines. The assignment (or placement) considers a set of hard and soft constraints, which are evaluated sequentially for each VM. Examples of hard constraints include not violating the physical capacity of a machine and not running a VM type on hardware that does not support it. An example of a soft constraint is to prefer an already-occupied machine to increase packing efficiency [21].

In this section, we discuss the general resource management problem in Azure by focusing first on challenges arising from both dynamic and diverse demand patterns (§2.1), as well as fluctuations in the available compute supply (§2.2). These challenges, coupled with the fact that demand might exceed supply, motivate the need for a robust admission-control system; we outline its requirements in §2.3.

### 2.1 Demand Versatility

Azure has multiple different offerings for compute, which each impose specific requirements on the underlying allocation system.

**VM requests.** Azure has multiple hardware generations in its datacenter and offers over a thousand VM types. The majority of VM types are supported on most hardware generations, but some types require specialized hardware (e.g., GPUs for ML). We note that other major cloud providers like AWS and GCP also offer a large number of VM types [5, 19].

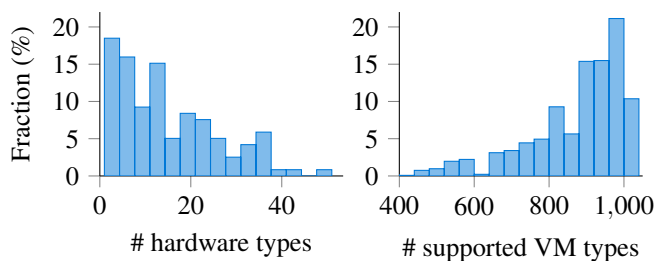


Figure 3: Histograms for number of different hardware types (left) and supported VM types (right) across Azure zones.

New VM types are regularly introduced as new hardware types and scenarios are onboarded to the cloud.

On-demand *VM requests* are the most common capacity consumption mode. A VM request specifies the type of the VM (which in turn determines the number of CPU cores, memory, disk, network requirements for the VM, and optional accelerators) and the VM’s priority. Multiple VM requests may be grouped into a *tenant request*. A tenant request is accompanied by a tenant service model, where additional constraints can be imposed on the collection of VM requests (e.g., fault-domain requirements).

By default, VMs are spread across an entire zone. However, a tenant may request all its VMs be co-located within specific inventory boundaries such as a cluster or datacenter. A tenant using legacy Azure services can also be pinned to a single cluster; this means that such a tenant cannot create new VMs outside its cluster.

A tenant request succeeds only if all its associated VM requests are successfully allocated. There is no explicit SLA on allocation time, but it is desirable to keep these times as low as possible to ensure a fast and reliable deployment experience. The volume of VM requests is large: a zone can handle more than two million requests in a day. The demand pattern can be quite bursty: Figure 4 shows that a zone can easily receive a few thousand requests per minute. Hence, low latency and high throughput are critical requirements for the VM allocation service (see §2.3).

Higher-order consumption modes, such as Function-as-a-Service (FaaS or serverless computation [39]) are internally provisioned through VMs.

**Customer scale-outs.** Customers may decide to increase the number of VMs in their tenants; these are termed *tenant scale-outs*. Any allocation request, including a scale-out request, must be handled within milliseconds; the system can either accept or reject the scale-out request based on available capacity. Though there is no external acceptance SLA for scale-outs, Azure internally tracks the acceptance rate of these requests and attempts to sustain very high acceptance rates (at least 4 nines). Towards this end, the admission-control logic must explicitly reserve capacity within individual clusters for scale-out of pinned tenants (see §3.1). We note that

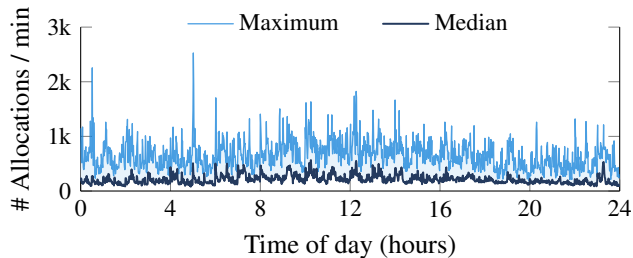


Figure 4: Number of high-priority allocations per minute for a zone over a day, aggregated over a 2-week period. Demand in the tail is bursty with large spikes.

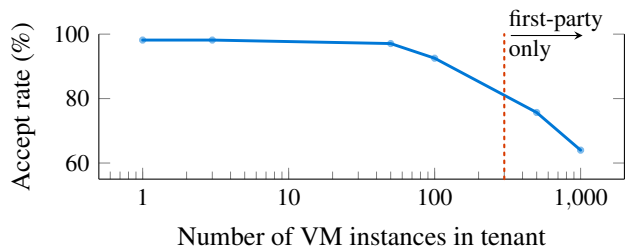


Figure 5: Tenant size versus allocator's acceptance rate. Data is obtained from one zone during a busy month. Tenant requests above 300 VMs are currently offered only internally.

Azure does not distinguish between scale-outs and new tenant requests for tenants that can span across an entire zone.

**Reservations.** Consider a scenario where a user wishes to terminate their VMs for the day, but expects to re-instantiate the VMs the following morning. Doing so through a regular on-demand tenant request might result in delays if capacity is not immediately available in the morning. As Figure 5 shows, this issue is exacerbated for large tenant requests. To address such scenarios, Azure offers the *reservation* option. A reservation request includes the VM type and the quantity. A reservation need not result in *actual* allocation of VMs; instead, it serves as a commitment from Azure to provide the desired number of VMs in the future, whenever the customer decides to materialize the reservation.

Reservations are active as soon as the request is accepted (i.e., we do not support reservation requests with future start dates). The user can terminate a reservation at any time. To support economy of scale, VMs that are created against a reservation cannot be pinned to a single cluster. This basic reservation model is offered by large public clouds as an *On-Demand Capacity Reservation* [4, 7]. We briefly discuss direct extensions of this basic reservation model, such as reservations with a future start date, in §6. Other reservation models are surveyed in §7.

## 2.2 Supply Fluctuations

Supply in Azure can change dynamically. In this subsection, we outline the situations that can trigger these changes.

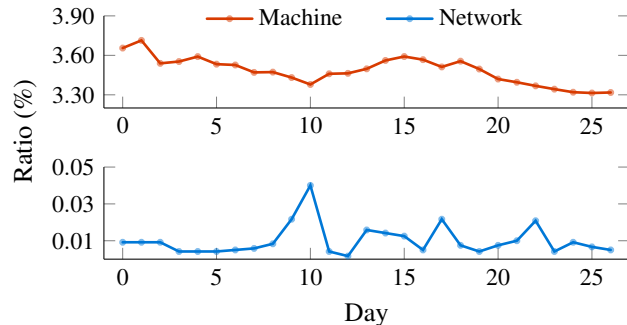


Figure 6: Network and machine failures across 30 days. The average network failure is around 0.01%, but each network failure can affect several machines. Outside of network failures, ~3.5% of machines are in a failed state every day.

**Machine and other hardware failures.** Figure 6 shows the failure frequencies for machines and network hardware over a period of 30 days. As the figure demonstrates, a network failure is much less probable. However, network failures impact more machines (e.g., a top-of-rack switch can affect around 50 machines). The cloud provider has to account for such failures and set aside capacity to migrate the VMs in affected machines to respect availability SLAs. It is thus crucial to both predict and have readily available mitigation for such failures.

**Additional events affecting supply.** Azure, like any other large cloud supplier, experiences other events that affect the available supply. One can roughly classify these events based on their predictability. *Deterministic* events include planned maintenance of machines (e.g., software updates) and decommissioning of hardware. *Stochastic* events include urgent security updates to patch a zero-day vulnerability, hotfixes to handle software bugs, and malicious attacks that overload the system.

## 2.3 Admission Control

We now describe the problem of admission control, and highlight requirements and design challenges for *Kerveros*. Admission control can be defined as the set of decisions that determine the acceptance of any form of resource request (new tenant, scale-out, reservation). Admission-control decisions need to take into account not only the present state of the zone, but also potential realizations of future state. Future state is influenced by a variety of factors including reservations that have been admitted but not yet materialized, potential hardware failures, tenant scale-outs, etc.

**Challenges.** As mentioned earlier, admission control in the cloud presents several challenges:

1. **Multiple elements affecting demand and supply.** *Kerveros* has to consider multiple elements with different characteristics: on the demand side, one has to consider

different modes of consumption (e.g., tenant vs. reservation requests) as well as potential tenant growth. On the supply side, we have different types of events (e.g., machine failures, rack maintenance) that affect inventory availability.

2. **Zone heterogeneity.** Beyond the inherent stochasticity in demand and supply, it is hard to determine how much capacity to set aside since not all machines can serve all VM types. For example, a new VM series might be served only on new hardware generations.
3. **Numerous VM sizes and fragmentation.** Unlike traditional resource allocation problems, which are often “single dimensional”, admission control for the cloud needs to make decisions for VMs that request different types of resources (e.g., CPU, memory, disk). These multiple dimensions can cause fragmentation.
4. **Accounting for unclaimed capacity.** Reservations and other types of capacity protection impose an additional challenge since we would ideally like to “late bind” resources to such requests. This makes machine utilization an imperfect metric to determine if enough capacity is available for a new request.

**Requirements.** An admission-control system also has to meet several requirements to operate effectively within the context of a large cloud provider:

1. **Scalability.** Even for very large zone inventories, the decision of whether to accept or reject a tenant or reservation request has to be made within milliseconds.
2. **Respect admitted reservations.** All VMs that are requested against a previously accepted reservation should be fulfilled (i.e., assigned to machines) whenever the user opts to materialize the reservation.
3. **Availability.** The cloud provider must ensure high availability to customers. Accordingly, capacity must be set aside to facilitate the migration of VMs to other machines in case of machine failures or other events affecting supply.
4. **Elasticity.** An admission-control system should reserve capacity to allow tenants pinned to clusters to scale out.
5. **Accuracy.** The system should not reject requests when capacity is in fact available.
6. **Efficiency.** The system should set aside as little capacity as possible while satisfying the above properties (i.e., increase the return on investment).

**Time scale of decision making.** We observe that the above challenges and requirements necessitate that decisions be made at different time scales. For example, estimating how much capacity to set aside for failures or growth requires comprehensive data analysis that is inherently time consuming. On the other hand, the VM allocation service itself has to remain highly performant and process requests at low latency. Consequently, a natural design choice is to decouple the overall admission-control responsibility between fast- and slow-twitch systems. Accordingly, the allocator only executes

low-overhead capacity limit checks to determine whether to accept or reject a resource request (i.e., admission control *enforcer*)<sup>1</sup>. The entire logic for estimating the available capacity is performed by *Kerveros*, the focus of this paper, and is performed off the critical path.

### 3 Design of *Kerveros*

The high-level goal of *Kerveros* is to answer the following question: *How much capacity is available for an incoming tenant or reservation request?* As described earlier, this information is used by the allocator to accept / reject requests.

To address the complexities of demand and supply fluctuations (§2), our approach relies on two main concepts: (i) *buffers*, to specify a need for capacity; and (ii) *allocable VM count*, an auxiliary bookkeeping technique to quickly determine the available capacity for an incoming tenant or reservation request.

#### 3.1 Buffers

A buffer is an accounting of capacity that must be protected for a specific purpose. The allocator treats this capacity as unavailable when admitting tenant or reservation requests. *Kerveros* supports three buffer types: (i) *Reservation buffers* to accommodate already admitted reservations; (ii) *Growth buffers* to accommodate existing tenant growth; (iii) *Healing buffers* to accommodate currently-running VMs that might need to be migrated in case of hardware failures.

As we detail below, we use appropriate counts of VM types to quantify the size of each buffer. Formally, a buffer is defined as a tuple  $(t, x)$ , where  $t$  is the VM type and  $x$  is the number of VMs of that type that ought to be protected. The size of each buffer may change over time. For instance, a reservation buffer can become smaller as the customer gradually starts using VMs corresponding to that reservation. On the other hand, a healing buffer may become larger over time, e.g., as hardware ages and failures become more likely.

We make an important design choice for buffers: although capacity is protected, it is *not* mapped to specific physical machines. Buffers are defined at higher levels of the cloud hierarchy (e.g., cluster or zone) to reflect the amount of capacity that must be protected in aggregate at that level. The physical allocation occurs only when the protected capacity is needed to fulfill its purpose. For example, after hardware failures, healing buffer capacity can be used for migrating VMs from the affected machines. To ensure maximum utilization of resources, we use unclaimed protected capacity to offer spot VMs [2,3], which can be immediately preempted to free up capacity whenever buffer capacity is claimed.

---

<sup>1</sup>For a tenant request, the capacity limit check is immediately followed by an actual allocation of the VMs to physical machines. For reservation requests, the allocator performs only the limit check since placement is late-bound for reservations in our design.



We next provide more details on how *Kerveros* sets sizes of buffers of different types.

**Reservation buffers.** Setting reservation buffers is straightforward: the buffer size is set exactly according to the user-provided reservation requirement. That is, suppose reservation  $k$  requires  $x$  VMs of type  $t$  within a certain zone; *Kerveros* sets the respective zone-level buffer as  $R_k = (t, x)$ . When the user claims  $u$  VMs of that reservation, the buffer is updated to  $R_k = (t, x - u)$ .

**Growth buffers.** Growth buffers are used at a cluster level to account for the expansion of existing tenants pinned to each cluster. *Kerveros* defines a single growth buffer for each VM type. Intuitively, the buffer size should be proportional to the current consumption of that VM type. More specifically, for a given cluster  $c$  and VM type  $t$ , let  $x_{tc}$  be the current number of active VMs belonging to tenants pinned to cluster  $c$ . Then we set the corresponding growth buffer to  $G_{tc} = (t, \alpha_{tc} x_{tc})$ , where  $\alpha_{tc} > 1$  is the effective growth rate. We use a ML model to set this parameter. In a nutshell, we consider a small set of possible effective growth rates (e.g., five values in the range between 1.03 and 1.1). The input features to the ML model consist of tenants' information (e.g., account IDs, VM lifetimes), hardware details (e.g., generation, SKU), and cluster-specific fragmentation details (intuitively, a cluster that is "badly" packed would induce higher effective growth rates); the statistics on fragmentation are obtained from the allocator. The ML model (implemented using XGBoost) is trained daily using historical growth data.

**Healing buffers.** *Kerveros* uses healing buffers to account for hardware failures and other events affecting supply (§2). For example, the protected capacity may be used to migrate VMs away from non-functional hardware. Since tenants can be constrained to specific clusters, healing buffers are maintained at the cluster level. Since full-machine VMs are the hardest to allocate (as they require a completely empty machine), the healing buffer is defined in units of full-machine VMs; formally, the buffer is of the form  $H_c = (L, x_c)$ , where  $L$  denotes the full-machine VM type, and  $x_c$  is the quantity.

We calculate  $x_c$  using a data-driven approach. The high-level idea is to set  $x_c$  in proportion to the hardware failure probability (i.e., the buffers should be larger if the failure probability is higher). More specifically, we would like to ensure that the total number of non-functional machines does not exceed the buffer size  $x_c$  with high probability; let  $p_c$  denote that probability (e.g.,  $p_c = 99.9\%$ ). Towards this end, we extract from historical failure data the empirical distribution of the total number of non-functional machines over a given period of time (e.g., a day). The total number accounts for a variety of events including machine and network failures as well as maintenance events. Then,  $x_c$  is simply derived as the  $p_c$ -percentile value of this distribution. *Kerveros* may add some slack to the obtained value (e.g., 10%) for clusters that are highly fragmented (as perceived

by the allocator). The choice of  $p_c$  is specific to the cluster, and depends on various factors. For example, *Kerveros* uses higher  $p_c$  values for clusters that have a high ratio of tenants pinned to that cluster, since such tenants have fewer fallback options in case of failures.

In rare cases, the healing buffer capacity is increased to allow completion of urgent software updates.

## 3.2 Allocable VMs

*Kerveros* uses the notion of allocable VM counts (AV counts) to reason about available capacity. Specifically, the AV count,  $\mathbb{A}[t]$ , gives the number of additional VMs of type  $t$  that can currently fit in the zone. For an incoming VM request of type  $t$  and demand  $x$ , *Kerveros* first calculates<sup>2</sup>  $\mathbb{A}[t]$ , and then the allocator rejects the request if  $\mathbb{A}[t] < x$ .

In this section, we describe how *Kerveros* calculates the AV counts across entire clusters and zones. The calculations are non-trivial because they require a *conversion* mechanism that accounts for buffers of different VM types  $t$  defined at different hierarchies of the datacenter (cluster versus zone). We note that other approaches that do not explicitly account for multiple resource dimensions might result in either under- or over-estimating the available capacity.

### 3.2.1 Overview

The high-level pseudocode of our AV count calculation is provided in Algorithm 1. Informally, the algorithm implements the following calculation:

$$\mathbb{A}[t] = \left[ \begin{array}{c} \text{available capacity for} \\ \text{type } t \text{ across clusters} \end{array} \right] - \left[ \begin{array}{c} \text{buffers converted from} \\ \text{type } t' \text{ to type } t \end{array} \right].$$

The algorithm starts by initializing the AV counts, excluding buffers, and accounting only for the active VMs in the system (Step 1). Since certain buffers are defined only at a zone level, the algorithm then proceeds to transform them to cluster-level buffers (Step 2), so that all buffers can be analyzed at the same level of the cloud hierarchy. The remaining calculations are done at a cluster level, except a final aggregation step. The heart of the algorithm is converting buffers of other VM types into buffers of type  $t$  (Step 7) and then deducting them from the current AV count (Step 8). This conversion step is the subtle part of the algorithm. We term the full algorithm the *Conversion Ratio Algorithm (CRA)*.

### 3.2.2 CRA Algorithmic Details

We next describe each of these steps in detail.

**AV count initialization (Step 1).** The initialization step calculates  $\mathbb{A}[t, c]$ : the number of VMs of type  $t$  that can fit in cluster  $c$ , excluding buffers. Each VM type requires different quantities of the various compute resources (e.g., CPU, memory) available in a machine; thus these non-buffered AV counts are obtained by calculating the maximum

<sup>2</sup>In practice, we update the AV counts only periodically (every minute), and use caching to track the updated AV counts. See §4 for details.

---

**Algorithm 1** CRA: Calculating AV Counts for a VM type  $t$ .

- 1: Calculate the AV counts excluding buffers:  $A[t', c]$  for all  $t'$ .
  - 2: Distribute zone-level buffers to clusters.
  - 3: **for** each cluster  $c$  **do** ▷ Calculate per-cluster counts.
  - 4:      $a_c = A[t, c]$
  - 5:     Aggregate buffers per VM type.
  - 6:     **for** each aggregate buffer  $(t', x)$  **do**
  - 7:         Convert buffer from  $t'$  into type  $t$ :  $\mathbb{C}(t' \rightarrow t, x, c)$ .
  - 8:          $a_c -= \mathbb{C}(t' \rightarrow t, x, c)$  ▷ Deduct from current total.
  - 9:     Aggregate cluster counts for zone:  $\mathbb{A}[t] = \sum_c a_c$ .
- 

number of VMs that can fit in each machine (considering all resource dimensions) and taking the sum over all machines in the cluster:

$$A[t, c] = \sum_{\substack{\text{machine in} \\ \text{cluster } c}} \min_d \left\lfloor \frac{\text{Available resource } d \text{ on machine}}{\text{Required resource } d \text{ for type } t \text{ on machine}} \right\rfloor.$$

**Going from zone- to cluster-level (Step 2).** *Kerveros* temporarily breaks down the zone-level reservation buffers into cluster-level buffers, so that all buffers can be analyzed at the cluster level. The apportioning to cluster-level buffers is done by mimicking how the allocator would spread the VMs across multiple clusters when reservations are materialized. Considerations that are taken into account include load balancing, prioritizing newer-generation hardware for new VM types, and more. We omit details for brevity.

**Aggregating cluster buffers per VM type (Step 5).** Each cluster may have multiple buffers of the same VM type. We aggregate all buffers of the same type into a single buffer by summing their sizes. Intuitively, having a single buffer makes the conversion across types (discussed next) less lossy and more efficient.

**Converting buffers into other VM types (Step 7).** We convert a buffer from one type ( $t'$ ) to another ( $t$ ) to estimate the impact that the original buffer has on allocations of type  $t$ .

One way to accomplish this is by using conversion ratios that encode the relative sizes of VM types  $t$  and  $t'$ . However, naïve conversion ratios have issues, since VMs have multi-dimensional sizes. To illustrate this, consider a simplified example with the following assumptions:

- Two requested resource types: CPUs and memory.
- Two VM types: large (2 CPUs, 4 GB) and small (1 CPU, 1 GB).
- Two machine sizes:  $M_1$  (25 CPUs, 40 GB) and  $M_2$  (25 CPUs, 25 GB).

Table 1 shows the counts of each VM type that can fit in one empty  $M_1$  and  $M_2$  machine, and also the counts after adding 10 and 20 small VMs using various counting methods. Simple conversion ratios based on resource ratios in isolation (CPU and RAM in Table 1) can lead to sub-optimal (yellow bars) or incorrect outcomes (red bars). This has repercussions on packing efficiency and SLA adherence: underestimates can

| Machine State | Method | AV Counts      |                |                        |    |
|---------------|--------|----------------|----------------|------------------------|----|
|               |        | Large on $M_1$ | Large on $M_2$ | Small on $M_1$ & $M_2$ |    |
| Empty         | Actual |                |                |                        |    |
| 10 small      | Actual |                |                |                        | 15 |
|               | CPU    |                |                |                        | 15 |
|               | RAM    |                |                |                        | 15 |
|               | CRA    |                |                |                        | 15 |
| 20 small      | Actual |                |                |                        | 5  |
|               | CPU    |                |                |                        | 5  |
|               | RAM    |                |                |                        | 5  |
|               | CRA    |                |                |                        | 5  |

Table 1: Counts of large and small VMs that can fit in two machines  $M_1$  and  $M_2$  using various other conversion-ratio-based methods compared to the true count (Actual). CPU treats 1 small VM as 1/2 a large VM, and RAM treats 1 small VM as 1/4 a large VM.

strand resources (leading to fragmentation and worse packing efficiency) and overestimates can violate hardware constraints (leading to SLA violations). The optimal conversion ratio depends both on the machine size and the VMs already allocated (i.e., resources *remaining* on the machine).

We use the ratio of AV counts as a low-dimensional approximation instead. Formally, a buffer of type  $t'$  with size  $x$  is converted to a buffer of type  $t$  with size:

$$\mathbb{C}(t' \rightarrow t, x, c) = \left\lfloor \frac{A[t, c]}{A[t', c]} \cdot x \right\rfloor.$$

This works well in practice (empirical results in §5).

**Example.** To illustrate CRA, we return to the example in Figure 1. As a quick recap, we consider a single cluster with two empty machines, each with size of 100 units (for simplicity, we assume only a single resource dimension). The three VM types small ( $S$ ), medium ( $M$ ), and large ( $L$ ) have sizes 20, 50, and 60 units respectively. A single medium-sized VM is requested – this request is rejected if  $\mathbb{A}[M] < 1$ . We first initialize the counts for our VM types ( $S$ ,  $M$  and  $L$ ):

$$A[S, c] = 2 \cdot \left\lfloor \frac{100}{20} \right\rfloor = 10, \quad A[M, c] = 2 \cdot \left\lfloor \frac{100}{50} \right\rfloor = 4, \\ A[L, c] = 2 \cdot \left\lfloor \frac{100}{60} \right\rfloor = 2.$$

In both scenarios in Figure 1, there is a single incoming request of size 50 and buffers of 120 units total that need to be taken into account; the only difference between the scenarios is the VM type of these buffers:

- *Large buffers.* Assume that we have two large buffers. We convert these buffers to type medium, which yields  $\mathbb{C}(L \rightarrow M, 2, c) = 4$  medium VMs. This results in  $\mathbb{A}[M] = 0 < 1$ , so we reject the request.

$$\mathbb{A}[M] = A[M, c] - \mathbb{C}(L \rightarrow M, 2, c) \\ = A[M, c] - \left\lfloor \frac{A[M, c]}{A[L, c]} \cdot 2 \right\rfloor = 4 - \left\lfloor \frac{4}{2} \cdot 2 \right\rfloor = 0.$$

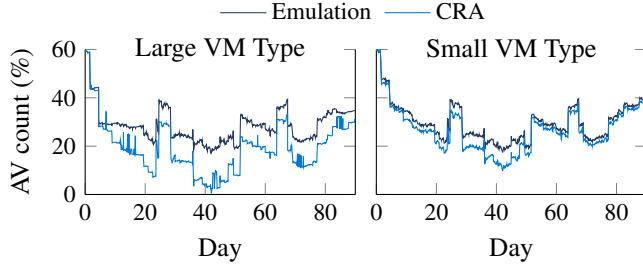


Figure 7: CRA versus emulation for two VM types, where the AV count is expressed as a percentage of the total capacity. Simulated results from a single trace.

- *Small buffers.* Assume that we have six small buffers. We convert these buffers to type medium, which yields  $\mathbb{C}(S \rightarrow M, 6, c) = 3$  medium VMs. This results in  $\mathbb{A}[M] = 1 \not\leq 1$ , so we accept the request.

$$\begin{aligned} \mathbb{A}[M] &= \mathbb{A}[M, c] - \mathbb{C}(S \rightarrow M, 6, c) \\ &= \mathbb{A}[M, c] - \left[ \frac{\mathbb{A}[M, c]}{\mathbb{A}[S, c]} \cdot 6 \right] = 4 - \left[ \frac{4}{10} \cdot 6 \right] = 1. \end{aligned}$$

In Appendix A, we provide a theoretical analysis of CRA under simplified assumptions; specifically, we prove that the conversion results in a bounded waste of resources.

### 3.2.3 Linear Adjustment Algorithm (LAA)

The Conversion Ratio Algorithm offers an efficient and scalable approach for obtaining the AV counts. Nevertheless, the output of this algorithm might sometimes be fairly inaccurate. The main reasons for inaccuracy are potential fragmentation issues while dealing with conversion between multi-dimensional VM types and not explicitly modeling how the VMs would be placed using the allocator (e.g., erroneously assuming the rightmost outcome in Figure 1 and rejecting the new request).

The above limitations can be mitigated if *Kerveros* could emulate the placing of the different buffers and filling up of the inventory by allocating VMs using the allocator. This is the main idea behind the *Linear Adjustment Algorithm (LAA)*. Since such emulation is compute-intensive and time-consuming, we run it periodically (every 30 minutes) and in isolation, i.e., without interfering with the handling of customer requests (more details in §4.1.2). We then use the emulation result to calibrate CRA’s output.

To see how the emulation output should be accounted for, we compare its output to CRA’s output. Figure 7 shows a time series of emulation and CRA’s output for two different VM types. We observe that the gap between the two methods is steady at times, but is spiky at other times. The LAA should account for both these phenomena.

Formally, for any VM type  $t$ , let  $\mathbb{A}'[t]$  and  $\mathbb{E}'[t]$  be the AV counts obtained by CRA and the allocator emulation at the time when the emulation was run last, respectively. We then

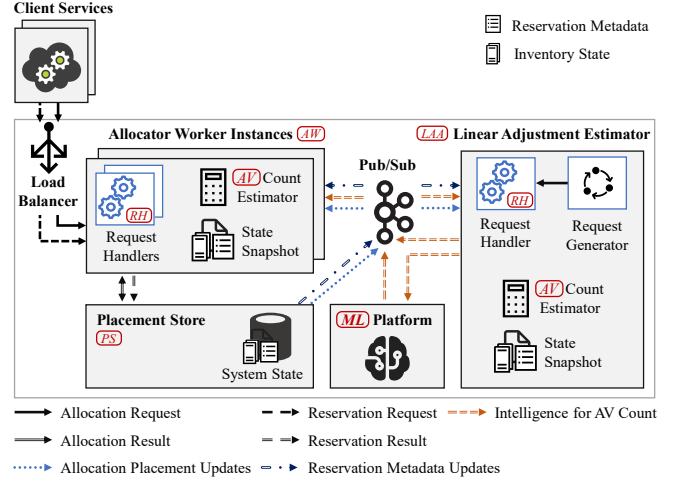


Figure 8: *Kerveros* system architecture.

adjust the current VM count estimate  $\mathbb{A}[t]$  (result of CRA) to obtain a new estimate:

$$\mathbb{A}_{\text{adjusted}}[t] := \beta_1' \cdot \mathbb{A}[t] + \beta_2' \cdot \mathbb{A}'[t] + \beta_3' \cdot \mathbb{E}'[t] + \beta_4', \quad (1)$$

where the coefficients  $\beta_i'$  are learnt using standard linear regression, with the emulation outputs serving as ground truth for training.  $\beta_2'$  and  $\beta_3'$  incorporate information on the gap between ground-truth and CRA in the previous time step, and  $\beta_4'$  models the constant gap between the two methods. In §5, we show that LAA substantially decreases AV count error, while maintaining scalability.

## 4 System Implementation

In this section, we first describe *Kerveros*’s various microservices that allow it to respond to allocation and reservation requests in an availability zone and enforce its admission-control logic in an efficient and scalable manner (§4.1). We then discuss tradeoffs that arise from various design decisions made in our implementation (§4.2).

### 4.1 Architecture

*Kerveros*’s microservices work in concert to handle allocation and reservation requests, estimate AV counts, execute emulation runs for AV count adjustment, persist state when allocations and reservations are accepted by the system, and train ML models as required (Figure 8). The microservices primarily use a distributed publish-subscribe (pub-sub) platform [42] to transfer state among themselves.

#### 4.1.1 Allocation Worker Instances

Multiple *stateless* allocation worker instances (AW) are used to handle both reservation and tenant allocation requests. An allocation worker instance is a process typically running on a dedicated machine. Each instance has two types of agents:

request-handling (RH) agents to serve requests, and an AV count estimation agent to periodically compute the number of allocable VMs. The number of worker instances and RH agents deployed in a zone is configured based on the request demand in the zone and size of the allocation inventory.

**Request handlers (RH).** A RH handles both reservation and tenant requests.

*Reservation requests.* For a reservation request, the RH performs a zonal admission-control check. The zonal check evaluates whether there is enough capacity in the zone by comparing the number of requested VMs with the zonal AV count. All buffer types (reservation, healing, and growth) are considered for this check. On success, the reservation metadata is persisted to the placement store (see below).

*Tenant requests.* For new tenant requests, the same zonal admission-control check is executed. The agent then proceeds to filter and sort the inventory machines for each requested VM, following a series of hard and soft constraints respectively, to assign a specific machine for each VM. One of the filtering steps involves performing cluster-scope admission-control checks to ensure clusters have sufficient capacity for the requested VMs.

The buffers used in the admission-control steps are adjusted based on the nature of the request and scope of the check. For example, at cluster scope, only healing and growth buffers are considered for new tenants when computing AV counts. Similarly, only healing buffers are considered for tenant growth requests. As another example, all admission-control checks are skipped for reservation-backed VM requests (i.e., requests for VMs against an already-accepted reservation) and healing requests, since these requests have already been accepted (either as part of the reservation or as part of the original VM / tenant request before hardware failure).

On success, the “VM → machine” mapping is persisted in the placement store.

**AV count estimator (AV).** The AV count estimator executes both CRA (§3.2.2) and the linear adjustments (§3.2.3). To avoid adding latency to the RH response time, the estimator is implemented as a separate agent off the critical path of the RH. It runs in a tight loop (every minute), and updates the RHs with new AV counts through in-memory state transfer.

To obtain the AV counts, the estimator requires information about machine occupancy and health, reservation metadata, adjustment coefficients, and reservation-VM maps for VMs allocated against reservations. Each estimator learns about this information through the pub-sub platform. This information is organized through multiple pub-sub topics. Given the distributed nature of the platform, each estimator works with a somewhat stale view of the inventory and reservations (we discuss the impacts of this in §4.2).

#### 4.1.2 LAA Instance

The LAA instance performs periodic emulation to obtain more accurate AV counts (§3.2.3). To do so, it listens to the same pub-sub topics as the worker instance, but does not handle customer requests and does not persist any results to the placement store. It starts each emulation run from a snapshot of the entire inventory (with buffer information). It then creates allocation requests corresponding to the buffers and allocates them using the snapshot of the inventory state. These results are stored as local in-memory modifications on the initial inventory snapshot. Once the buffers are allocated, the LAA instance computes more accurate AV counts for each VM type from the remaining available capacity by repeatedly allocating (till failure) and deallocating VMs for each type. We note that these operations do not interfere with the critical path of real request handling or admission-control enforcement. Finally, the LAA instance sends relevant estimation data to the ML platform, which runs the linear regression required to tune the  $\beta_i$  parameters in Equation 1; the training is performed at a coarser time granularity (every day, using a week’s worth of emulations results).

#### 4.1.3 Offline ML Platform

The offline ML platform (ML) performs relevant ML training tasks; its output is consumed by the AV count estimators. In addition to updating the LAA coefficients, the platform provides the predictions required for buffer management (e.g., determining the effective growth rate for a cluster).

#### 4.1.4 Placement Store

A *stateful* microservice, called the placement store (PS), persists the results of the computations performed by the RH agents in the worker instances.

Correctness of the VM→machine assignment is critical since incorrect assignments can lead to VM start failures and violation of explicit guarantees provided to customers. Hence, the PS validates each VM→machine assignment to check for conflicting assignments made by other concurrent RH agents, and returns a retry if validation is not successful. This ensures that the RH’s stale view of the inventory does not lead to correctness issues. PS uses fine-grained machine-level locking to allow results from multiple machines to be checked and persisted concurrently.

On the other hand, validating admission-control checks at a zonal level with concurrent allocation workers would require the PS to take a global lock on the entire inventory. This would severely compromise scalability. Hence, the PS does not perform any *global* capacity checks, which means that buffer enforcement is not guaranteed to be correct for every request. We note however that temporary reductions in buffer capacity are rarely seen in practice and are more acceptable (see discussion below).



## 4.2 Practical Considerations

We now briefly discuss the implications of some of these design decisions.

**Dependence on underlying allocator.** While we build and design *Kerveros* within the context of Azure, we note that other major public clouds face similar challenges (e.g., a multitude of VM types and consumption modes, hardware failures, efficiency requirements). Moreover, *Kerveros*'s algorithms are agnostic to the details of the underlying allocator (e.g., exact hard and soft constraints enforced).

**Caching.** As described in §3.2, AV counts are computed for every machine separately and then aggregated to derive the allocable VMs that can fit in the cluster. Instead of computing AVs for every machine from scratch, the AV counts are maintained in an in-memory cache per machine and per cluster. This cache is initialized when the AV count estimator starts, and then is updated incrementally only for the machines that are modified because of changes in machine occupancy or health. Caching the AV counts in this manner massively reduces the computation overhead because the number of machines altered between consecutive AV count updates is orders of magnitude smaller than the total number of machines in the inventory. The periodic nature of AV count computation (as opposed to calculating AV counts for every request) might lead to using protected capacity.

**Optimistic concurrency.** As noted earlier, *Kerveros* uses optimistic concurrency control for better scalability, which allows allocation worker instances to see stale versions of state in the system. This is a limitation, since it allows the possibility of accepting multiple requests eating into the "same" protected capacity. This can occur, for example, when multiple worker instances accept requests or reservations simultaneously while not being aware of each other.

Due to dynamic churn (in particular VM shutdowns) in the workload (Figure 9), usages of protected capacity are temporary and rarely result in SLA violations. Running out of capacity due to stale AV counts or optimistic concurrency is rare, but if it happens, we have multiple knobs for mitigation, such as temporarily eating into healing buffers, evicting internal non-critical workloads, restricting the creation of new VMs, and migrating VMs to reduce fragmentation.

## 5 Evaluation

In this section, we seek to answer the following questions:

- Does *Kerveros* perform better than relevant baselines on the following dimensions: packing efficiency, SLA violations, and runtime?
- How well do *Kerveros*'s various optimizations work to count AVs accurately (e.g., LAA)?
- How does *Kerveros* trade off accuracy vs. compute effort?

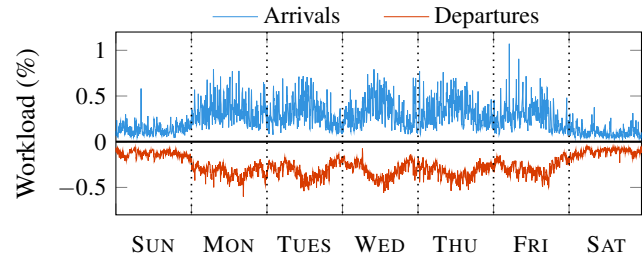


Figure 9: *Churn* for high-priority workloads in a single zone. Average is computed over 5-minute intervals, and computed over a 2-week period. Workload is weighted by the size of VM requested and normalized by the total volume of the zone. That is, roughly an average of 0.3% of total workload arrives and exits the zone every 5 minutes.

## 5.1 Experimental Setup

In this section, we outline the metrics of comparison, the baselines, as well as details on how we run experiments in production and simulation.

**Metrics.** We use the following metrics to measure *Kerveros*'s effectiveness:

- **Packing efficiency.** This metric estimates the capacity wasted by resource fragmentation due to inefficient packing. To measure packing efficiency, we temporarily fill the system with full-node VM requests, and set packing efficiency to be the final system load (including unused buffers) as a percentage of total cores (proxy). We choose full-node VM requests since they are generally the hardest to place.
- **Scalability.** The runtime (latency) of *Kerveros*'s AV count estimator.
- **SLA violations.** The proportion of requests where SLAs (machine failures, growth, reservation) are violated.
- **AV error.** The deviation in AV counts between offline emulation (which serves as an oracle) and other approaches. We use this metric to study the effectiveness of *Kerveros*'s approximate AV counting approaches.

**Baselines.** We compare *Kerveros* against the following:

- **Offline emulation (Oracle).** In the event of a tenant or reservation request, the allocator first makes temporary allocations for all unused buffers in the system. The allocator then checks the feasibility of accepting the current request; if there are not enough resources to satisfy the request, it is rejected. All temporary allocations for unused buffers are then discarded. This is slow but gives an accurate AV count, which can then be used as ground truth to estimate the AV error described above.
- **Placeholder (PH).** All buffers are allocated and assigned to physical machines at the time of request (or update). For example, an accepted reservation is allocated all of its required resources at admission time.

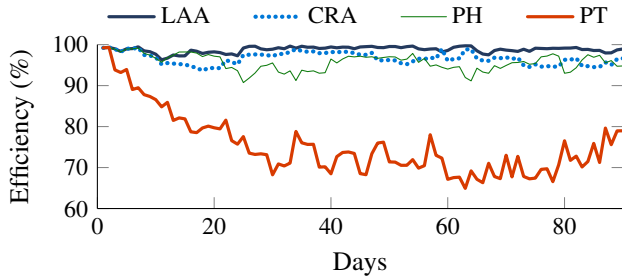


Figure 10: Comparison of packing efficiency across various baselines against the inventory partitioning algorithm (PT). The results are computed over a single trace. As can be seen, PT does not work well with our workloads.

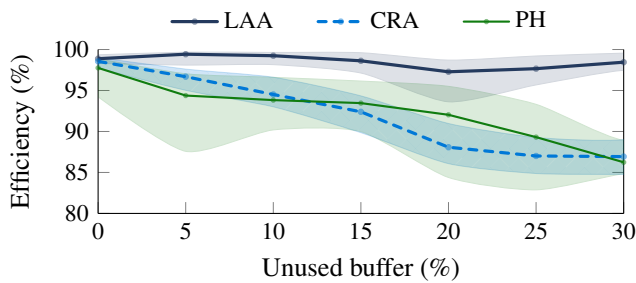


Figure 11: Packing efficiency vs. unused buffer size. We show the median and 25/75<sup>th</sup> percentiles, aggregated over all traces.

- **Inventory partitioning (PT).** Each buffer is allocated its own set of dedicated machines, enough to satisfy demand. The partitioning approach is similar to the method used by RAS [34], where *machines* are dynamically partitioned as new reservations arrive.

**Production.** Azure collects telemetry on different aspects of VM resource management in an internal data analytics platform. We use this data to measure healing and growth failures as well as latencies (§5.2). All production metrics are gathered for a period of one month in 2022.

**Simulator.** Production data is not sufficient to provide a full evaluation of *Kerveros*. In particular, comparison against other approaches requires a simulator since the baselines we consider are not deployed in our production setting.

To this end, we use our event-driven simulator to test various aspects of the entire admission control system (including both the allocator and *Kerveros*). Due to the allocator’s relative complexity, the simulator includes only a lightweight version, which supports the key constraints of the allocator logic. The simulator handles both tenant and reservation events (arrival, departure, and updates). Despite supporting a subset of the allocator’s logic, the simulator provides an excellent approximation of the system in production and is able to scale adequately to large inventories. We have extended the simulator to support the above baselines.

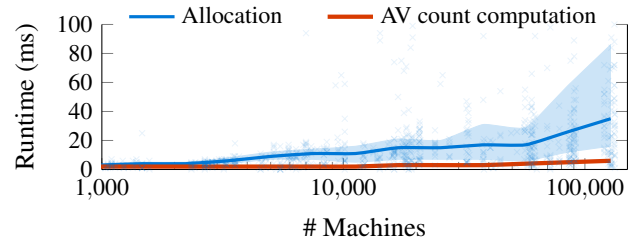


Figure 12: Latency (25/50/75<sup>th</sup> percentiles) versus inventory size, extracted from Azure over a month. Randomly-sampled allocation times are included for further context. End-to-end allocation latency is more significantly affected by inventory size compared to the AV count computations.

**Simulation traces.** We run simulation experiments on twenty traces that include both tenant and reservation requests. The traces use historical production data for tenant requests. Reservations, on the other hand, are a relatively new offering without significant traffic yet. Consequently, we synthetically add reservations to augment the historical data. The reservation characteristics, such as VM type and size, are extrapolated from real reservations.

## 5.2 End-to-End Experiments

Our goal here is to show that *Kerveros*’s CRA and LAA approaches outperform other baselines along three axes: packing efficiency, scalability, and SLA adherence.

### 5.2.1 Packing Efficiency

Figure 10 shows a timeline view of packing efficiency for one of the traces. The partitioning (PT) approach of splitting by machine is inefficient for our workloads: PT might work well in other limited settings (e.g., small number of total tenants). However, our general setting includes a large number of requests which require many fractions of machines. We therefore exclude PT from the rest of the analysis.

We now further explore the remaining baselines and aggregate results over all traces. Figure 11 illustrates how unused buffer sizes influence packing efficiency. As the unused buffer size increases, CRA suffers from accumulated rounding errors, and the PH algorithm can lock into sub-optimal packing decisions, resulting in inferior packing efficiency. LAA can compensate for rounding errors, and sustains high packing efficiency even with large unused buffers.

### 5.2.2 Scalability

Figure 12 shows the latency of the AV count computation for various inventory sizes. For reference, we also show the allocator’s latency for a single VM. We observe that *Kerveros*’s approximate AV counting algorithm scales well with inventory size, taking less than ten milliseconds even for inventories of over a hundred thousand machines. Approximate AV counting is cheap because the underlying computation is proportional to the number of VM types; in

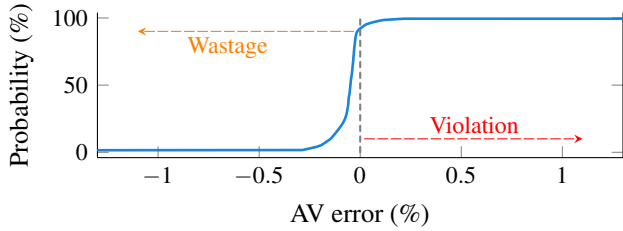


Figure 13: Cumulative density function (CDF) of AV error from LAA. SLA violations can occur when the AV error is positive (i.e., the algorithm overestimates AV counts, or asserts that there is more available capacity than reality).

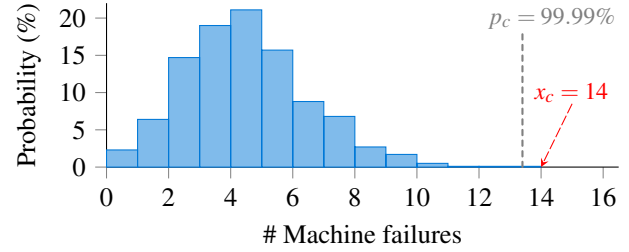
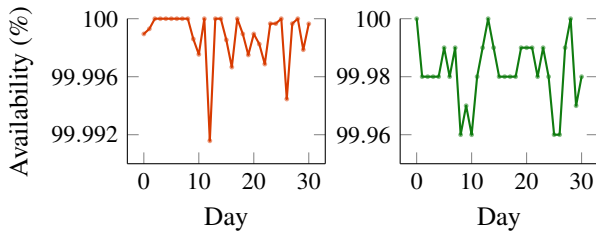


Figure 15: Illustration of healing buffer size, where  $x_c = \lceil 13.4 \rceil = 14$  machine-equivalent buffers are prepared to satisfy the desired 99.99% SLA.



(a) Healing.

(b) Growth.

Figure 14: The impact on availability due to healing and growth failures. Note that these values are not weighted by time, but represent a normalized count of all failures within each day. The actual loss of availability to each customer is less than what is indicated above. This historical data is aggregated over the entire cloud for a month in 2022.

contrast, the allocator runtime depends on the inventory size (since the allocator has to choose the most suitable machine for allocation). Overall, the results indicate that *Kerveros*'s AV-based approach is not the computational bottleneck.

### 5.2.3 SLA Violations

We also want to verify that *Kerveros*'s approach results in a low number of SLA violations. We verify this in both production and simulation.

**Reservation.** Figure 13 shows the cumulative density function (CDF) of AV errors obtained from simulation. When the AV error is positive, *Kerveros* overestimates the available capacity. This behavior might lead to reservation SLA violations in some extreme cases (for example, if all users' reservations, healing buffers, and growth buffers are claimed almost simultaneously). As the LAA curve shows, *Kerveros* reserves sufficient capacity to cover all promised resources around 86% of the time; *Kerveros* requires less than 1% additional capacity to satisfy requests about 99.7% of the time. With typical workload churn, we expect to obtain this additional capacity promptly.

**Healing.** Figure 14a shows production data over a month for instantaneous SLA violations due to healing failures.

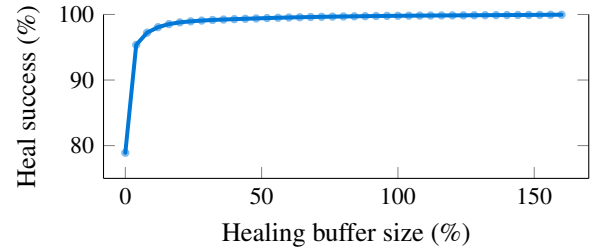


Figure 16: Tradeoff between healing buffer size and healing success rate.

We observe that *Kerveros* is able to sustain four nines of availability through the entire month by appropriately sizing healing buffers.

As Figure 15 shows, we calculate the healing buffer size based on a data-driven method (see §3.1). To better understand the tradeoff between packing efficiency and SLA adherence quantitatively for healing, we show how the healing buffer size influences the healing success rate based on data over a month in 2022, as shown in Figure 16. The  $x$ -axis shows the ratio of the buffer size compared to the size used in production, where 100% healing buffer size corresponds to the current production's decisions. When the ratio is zero, *Kerveros* does not reserve any healing buffers, but can still heal VMs affected by hardware failures if enough resources are coincidentally available in the cluster.

**Growth.** Figure 14b shows production data on instantaneous growth failures for the same month. We observe that growth is supported more than 99.9% of the time.

## 5.3 Deep Dive on AV Counting Algorithms

We now further evaluate the various components used to calculate AV counts.

**Improvements from linear adjustment.** Figure 17 shows the AV error of CRA (top) and LAA (bottom) versus the size of unused buffers. The estimation error with CRA increases with the unused buffer size, as fragmentation is amplified. LAA is more robust by fixing biases periodically (§3.2.3).

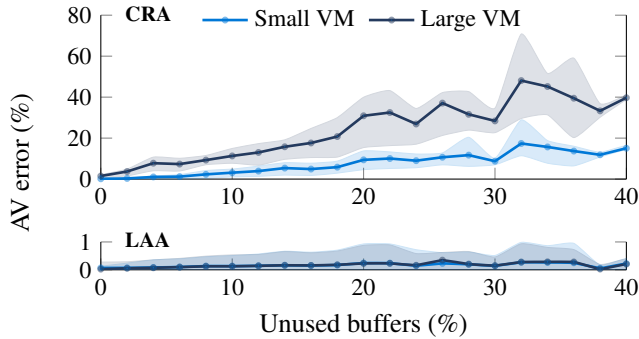


Figure 17: AV errors when calculating AV counts for two VM types (small and large) versus unused buffer size. The top graph, with average and 25/75<sup>th</sup> percentiles for CRA, shows CRA performing increasingly worse as unused buffer grows. LAA with an adjustment frequency of 30 minutes obtains small error even as the unused buffer percentage becomes large. We show 95<sup>th</sup> percentiles for LAA for emphasis.

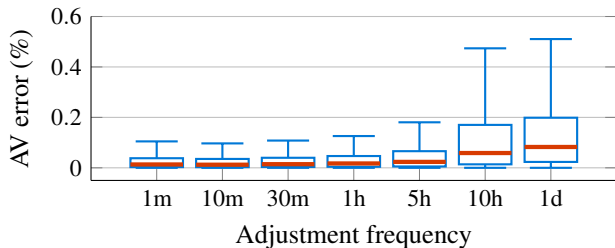


Figure 18: AV errors versus adjustment frequency in LAA. As we increase the frequency of adjustment, error is reduced at the expense of additional computation. In the  $x$ -axis, ‘m’ stands for minute, ‘h’ for hour, and ‘d’ for day.

**Adjustment frequency.** Figure 18 shows how the AV error changes with the frequency of LAA emulation. The figure shows that emulating every few minutes does not drastically reduce AV error. Consequently, we choose an emulation frequency of 30 minutes. Interestingly, we find that even with just one adjustment a day, the LAA algorithm still performs better than the basic CRA approach.

## 6 Discussion

We briefly discuss additional considerations relevant to *Kerveros*.

**The public cloud eco-system.** Modern public clouds have a large number of components that interact with each other, making multi-faceted decisions related to economically incentivizing usage, managing quota, provisioning capacity, etc. For example, special permissions or quotas are needed for very large customer demands; similarly, discounts are often offered to incentivize prominent customers. Such decisions are made on a slower time scale, often involving humans in

the loop, and influence the inputs to *Kerveros*.

**Constraints beyond capacity checks.** The decision of admitting a tenant request is complex, and involves a variety of constraints beyond capacity checks (e.g., fault domains, locality). However, when *Kerveros* reserves buffers (healing, growth, reservations), it can afford to take a simplified view, relying on the economy of scale (large inventory, workload churn) and mitigation actions as a last resort. Accordingly, the basic CRA algorithm does not account for the entire set of constraints. Nonetheless, CRA is supplemented by LAA. LAA in turn strongly relies on emulating the allocator’s logic, which does account for multiple preferences and constraints and uses realistic tenant requests against unused reservations. We note that the actual tenant requests against reservations can only specify a limited set of constraints by design (e.g., limit the maximum number of fault domains). The net effect is that *Kerveros* has proven to be reliable in production, despite the simplifying assumptions.

**Priorities.** While some business priorities are enforced at higher layers, allocation requests to *Kerveros* can have different priorities based on preemption level: higher-priority requests can preempt lower-priority ones to grab capacity [21, 45]. However, all buffers (reservation, growth, healing) are maintained only for the highest (non-preemptible) priority. Regardless of their priority, *Kerveros* handles all tenant and reservation requests on a first-come-first-serve basis.

**Predictive modeling.** Having a large percentage of unused buffers impacts resource efficiency; running preemptible workloads (e.g., spot VMs) on unused buffers is a partial mitigation. Currently, we avoid setting aside capacity in anticipation for future new customers. Incorporating ML models into *Kerveros* to predict resource usage and utilize the unused capacity even more aggressively is an interesting future direction.

**Comparison to early binding.** As an alternative to *Kerveros*’s late-binding approach, an early-binding or placeholder approach that exploits the existing allocation system to allocate buffers when needed (e.g., when a reservation request is accepted, or when a machine failure occurs) may seem like a more natural solution. Our evaluation of *Kerveros* against the baseline placeholder solution (PH) in §5 showed that this strategy suffers from worse packing efficiency. In addition, it introduces other various complications:

- Early binding requires lock-in of VM configuration parameters (e.g., VM type) when not necessary yet, reducing flexibility on the clients’ side.
- Buffers like the healing buffer need to have their sizes updated regularly, which is more inefficient with the early-binding approach.
- Early binding makes it harder to support over-subscription using spot VMs or harvest VMs.



- Early binding might require complex migration logic to move “placeholder” VMs between machines to pack VMs efficiently on the available physical machines.
- The early-binding approach incurs an additional caching cost, even if live migrations are free, i.e., the corresponding caches that reference the source and destination machines for each placeholder migration need to be invalidated.

**Reservations with future start dates.** *Kerveros* only supports reservations starting immediately. To guarantee reservations starting at future time  $t$ , we either need accurate estimates of the capacity at  $t$  (high risk), or we can reserve capacity now and hold it until  $t$  (high cost). Both approaches have significant issues that become worse as  $t$  is pushed out further into the future; these issues are also exacerbated by larger reservations (both in terms of the number and size of the reservation). It might be possible to estimate future capacity using a probability distribution, but we expect the accuracy to be poor without a reservation end time. Alternatively, rather than requesting a specific start and end time for a reservation, a user could specify a reservation with a *demand profile* that indicates the desired usage and expected growth over time.

## 7 Related Work

*Kerveros* builds on a rich line of previous work on admission control and reservations in the cloud.

**Admission control.** Admission control is a broad topic that has been studied in a variety of contexts such as cloud systems [9, 16, 26, 27, 29, 43, 45, 51, 52, 55], computer networks [8, 11, 18, 28, 30, 36, 44, 46, 47, 53], cellular networks [37, 50], mobile edge computing [1, 23, 24], real-time database systems [12, 22, 32, 38, 48], distributed systems [13–15, 41, 49, 54], and caching systems [10, 33, 40], among others. Each domain provides unique challenges and requirements that advocate for custom solutions to be developed. This work focuses on datacenter-scale VM admission control with support for VM reservations. To the best of our knowledge, there is no published work in this space that explicitly covers availability, scalability, and efficiency considerations. The bulk of the related papers in this space is centered more around VM allocation and placement, and admission control is addressed only at a high level. Similarly, reservations are a relatively new offering and are not directly addressed. For example, Protean [21] describes Microsoft’s rule-based zonal allocator, while focusing on systems enhancements to reduce latencies. However, Protean does not address either the problems of admission control or support for reservations. Google’s Borg [43, 45] scheduler introduces efficient packing and machine-sharing techniques to achieve high resource utilization, but admission control is only briefly described in terms of a quota system, and managing the fragmentation and quota allocation of the admission control is outside the scope of the paper. In terms of reservations, Borg uses the concept

of Borg Allocs, which are akin to the placeholder approach that we compare against.

**Reserving cloud resources.** The (VM) reservations we consider in this paper are a relatively new consumption mode that has been introduced to provide predictability to customers. Note that this mode is different from reserved instances (RIs) [6, 7, 17, 20, 35], under which customers make a 1-3 year *commitment* in return for a significant discount over on-demand offerings. For reservations in datacenters, we are only aware of Meta’s RAS [34] system, which manages user reservations at the granularity of a server. RAS works by partitioning machines and dynamically migrating machines between partitions, guided by a mixed-integer linear program. While the partitioning approach works well for Meta’s internal services, it is not well-suited for our public cloud setting (§5).

Resource reservations have also been proposed for internal big data analytics systems (e.g., [16, 25]). However, the provider there has more information about the jobs (e.g., job duration and deadline) and can better create an explicit resource allocation plan over time.

## 8 Conclusion

This paper describes the design, implementation, and evaluation of *Kerveros*, an admission-control system deployed in a large public cloud. Our design accurately retains capacity for hardware failures, tenant scale-out, and reservations while being computationally scalable to large inventories and peak loads. *Kerveros* can be extended to support additional scenarios such as improving margin efficiency through reservation overbooking and supporting enhanced reservation semantics (e.g., reservations with a start date).

## Acknowledgements

We thank our shepherd David Richardson and the OSDI reviewers for their valuable feedback. We also thank Bertus Greeff, Saurabh Agarwal and Ricardo Bianchini for useful discussions. The research of Timothy Zhu is supported in part by the National Science Foundation under Grant No. 1909004.

## References

- [1] Nadine Abbas, Wissam Fawaz, Sanaa Sharafeddine, Azzam Mourad, and Chadi Abou-Rjeily. SVM-based Task Admission Control and Computation Offloading using Lyapunov Optimization in Heterogeneous MEC Network. *IEEE Transactions on Network and Service Management*, 19(3):3121–3135, 2022.
- [2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–20, 2013.

- [3] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.
- [4] AWS. EC2 Capacity Reservations. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-capacity-reservations.html>, 2023.
- [5] AWS. EC2 Instance Offerings, 2023. <https://instances.vantage.sh>.
- [6] AWS. EC2 Reserved Instances. <https://aws.amazon.com/ec2/pricing/reserved-instances/>, 2023.
- [7] Microsoft Azure. On-demand Capacity Reservations. <https://azure.microsoft.com/en-us/reservations/>, 2023.
- [8] Sihem Bakri, Bouziane Brik, and Adlen Ksentini. On Using Reinforcement Learning for Network Slice Admission Control in 5G: Offline vs. Online. *International Journal of Communication Systems*, 34(7):e4757, 2021.
- [9] Gaurav Baranwal and Deo Prakash Vidyarthi. Admission Control in Cloud Computing using Game Theory. *The Journal of Supercomputing*, 72(1):317–346, 2016.
- [10] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.
- [11] Pablo Caballero, Albert Banchs, Gustavo De Veciana, Xavier Costa-Pérez, and Arturo Azcorra. Network Slicing for Guaranteed Rate Services: Admission Control and Resource Allocation Games. *IEEE Transactions on Wireless Communications*, 17(10):6419–6432, 2018.
- [12] M Carey, Sanjay Krishnamurthi, and Miron Livny. Load Control for Locking: The Half-and-Half Approach. In *Proc. of 9th Symp. on Principles of Database Systems*, 1990.
- [13] Huamin Chen and Prasant Mohapatra. Session-based Overload Control in QoS-Aware Web Servers. In *Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 516–524. IEEE, 2002.
- [14] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the 10th international conference on World Wide Web*, pages 545–554, 2001.
- [15] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.
- [16] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-Based Scheduling: If You’re Late Don’t Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.
- [17] Alibaba Cloud ECS. How to Use Alibaba Cloud Reserved Instances to Reduce Costs. [https://www.alibabacloud.com/blog/how-to-use-alibaba-cloud-reserved-instances-to-reduce-costs\\_595237/](https://www.alibabacloud.com/blog/how-to-use-alibaba-cloud-reserved-instances-to-reduce-costs_595237/), 2023.
- [18] Domenico Ferrari and Dinesh C Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, 1990.
- [19] GCP. Compute Engine Instance Offerings, 2023. <https://gcpinstances.doit-intl.com>.
- [20] GCP. Reservations of Compute Engine Zonal Resources. <https://cloud.google.com/compute/docs/instances/reserving-zonal-resources>, 2023.
- [21] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.
- [22] Hans-Ulrich Heiss and Roger Wagner. *Adaptive Load Control in Transaction Processing Systems*. Universität Karlsruhe. Fakultät für Informatik, 1991.
- [23] Dinh Thai Hoang, Dusit Niyato, and Ping Wang. Optimal Admission Control Policy for Mobile Cloud Computing Hotspot with Cloudlet. In *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 3145–3149. IEEE, 2012.
- [24] Jiwei Huang, Bofeng Lv, Yuan Wu, Ying Chen, and Xuemin Shen. Dynamic Admission Control and Resource Allocation for Mobile Edge Computing Enabled Small Cell Network. *IEEE Transactions on Vehicular Technology*, 71(2):1964–1973, 2021.
- [25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.
- [26] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora Varvarigou. Admission Control for Elastic Cloud Services. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 41–48. IEEE, 2012.
- [27] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora A Varvarigou. Elastic

- Admission Control for Federated Cloud Services. *IEEE Transactions on Cloud Computing*, 2(3):348–361, 2014.
- [28] J-Y Le Boudec. Application of Network Calculus to Guaranteed Service Networks. *IEEE Transactions on Information Theory*, 44(3):1087–1096, 1998.
- [29] Nikolaos Leontiou, Dimitrios Dechouniotis, and Spyros Denazis. Adaptive Admission Control of Distributed Cloud Services. In *2010 International Conference on Network and Service Management*, pages 318–321. IEEE, 2010.
- [30] Jörg Liebeherr, Dallas E Wrege, and Domenico Ferrari. Exact Admission Control for Networks with a Bounded Delay Service. *IEEE/ACM Transactions on Networking*, 4(6):885–901, 1996.
- [31] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, 2012.
- [32] Axel Mönkeberg and Gerhard Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data Contention Thrashing. In *VLDB*, volume 92, pages 432–443, 1992.
- [33] Giovanni Neglia, Damiano Carra, Mingdong Feng, Vaishnav Janardhan, Pietro Michiardi, and Dimitra Tsigkari. Access-Time-Aware Cache Algorithms. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 2(4):1–29, 2017.
- [34] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 505–520. Association for Computing Machinery, New York, NY, USA, October 2021.
- [35] Oracle. Capacity Reservations. <https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/reserve-capacity.htm>, 2023.
- [36] Josep Xavier Salvat, Lanfranco Zanzi, Andres Garcia-Saavedra, Vincenzo Sciancalepore, and Xavier Costa-Perez. Overbooking Network Slices through Yield-Driven End-to-End Orchestration. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 353–365, 2018.
- [37] Amilcare Francesco Santamaria and Andrea Lupia. A New Call Admission Control Scheme based on Pattern Prediction for Mobile Wireless Cellular Networks. In *2015 Wireless Telecommunications Symposium (WTS)*, pages 1–6. IEEE, 2015.
- [38] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *22nd International Conference on Data Engineering (ICDE’06)*, pages 60–60. IEEE, 2006.
- [39] Mohammad Shahradd, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Riccardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.
- [40] David Starobinski and David Tse. Probabilistic Methods for Web Caching. *Performance Evaluation*, 46(2-3):125–137, 2001.
- [41] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed Resource Management across Process Boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 611–623, 2017.
- [42] Khin Me Me Thein. Apache Kafka: Next Generation Distributed Messaging System. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.
- [43] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.
- [44] Guillaume Urvoy-Keller, Gérard Hébuterne, and Yves Dallery. Traffic Engineering in a Multipoint-to-Point Network. *IEEE Journal on Selected Areas in Communications*, 20(4):834–849, 2002.
- [45] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.
- [46] Matteo Vincenzi, Elena Lopez-Aguilera, and Eduard Garcia-Villegas. Timely Admission Control for Network Slicing in 5G with Machine Learning. *IEEE Access*, 9:127595–127610, 2021.
- [47] Boqian Wang, Zhonghai Lu, and Shenggang Chen. ANN-Based Admission Control for On-Chip Networks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.
- [48] Donghui Wang, Peng Cai, Weining Qian, and Aoying Zhou. Discriminative Admission Control for Shared-Everything Database under Mixed OLTP Workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 780–791. IEEE, 2021.
- [49] Matt Welsh and David Culler. Adaptive Overload Control for Busy Internet Servers. In *4th USENIX Sym-*

*posium on Internet Technologies and Systems (USITS 03)*, 2003.

- [50] Chen-Feng Wu, Liang-Teh Lee, Hung-Yuan Chang, and Der-Fu Tao. A Novel Call Admission Control Policy using Mobility Prediction and Throttle Mechanism for Supporting QoS in Wireless Cellular Networks. *Journal of Control Science and Engineering*, 2011, 2011.
- [51] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based Admission Control for a Software-as-a-Service Provider in Cloud Computing Environments. *Journal of Computer and System Sciences*, 78(5):1280–1299, 2012.
- [52] Haitao Yuan, Jing Bi, Wei Tan, and Bo Hu Li. CAWSAC: Cost-Aware Workload Scheduling and Admission Control for Distributed Cloud Data Centers. *IEEE Transactions on Automation Science and Engineering*, 13(2):976–985, 2015.
- [53] Xiaolu Zhang, Demin Li, Wei W Li, and Wei Zhao. An Optimal Dynamic Admission Control Policy and Upper Bound Analysis in Wireless Sensor Networks. *IEEE Access*, 7:53314–53329, 2019.
- [54] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.
- [55] Timothy Zhu, Daniel S Berger, and Mor Harchol-Balter. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 374–387, 2016.



# Appendix

## A Theoretical Guarantees

In this section, we show that the conversion ratio approach does not lead to an excessive wastage of resources.

**Notation.** We start by defining some useful notation.

- Let  $y_m^t$  denote the type- $t$  VMs that can fit in a machine  $m$ , ignoring buffers.
- Let  $x_c^t$  denote the number of VMs of type  $t$  required for buffers in cluster  $c$ , and let  $x_c^{t' \rightarrow t} = \mathbb{C}(t' \rightarrow t, x_c^t, c)$  be the converted number of VMs from type  $t'$  to type  $t$ .
- We define  $C_{t' \rightarrow t} = \frac{A[t, c]}{A[t', c]}$ , which allows us to write the conversion ratio from type  $t'$  to type  $t$  as

$$x_c^{t' \rightarrow t} = \mathbb{C}(t' \rightarrow t, x_c^t, c) = \lceil C_{t' \rightarrow t} \cdot x_c^t \rceil.$$

**Setting.** Our analysis focuses on the following setting:

- We assume that one of the resources (e.g., CPU) is the main bottleneck in a cluster; let  $q_t$  denote the amount of that resource that VM type  $t$  requires.
- When  $C_{t' \rightarrow t} < 1$  (i.e., we are converting from “smaller” to “larger” VMs), we omit the ceiling function from the converted AV count calculation. Instead,  $x_c^{t' \rightarrow t} = C_{t' \rightarrow t} \cdot x_c^t$ .
- Let  $M_c^{t, t'}$  denote the set of machines in cluster  $c$  that can currently fit both VM types  $t'$  and  $t$ . The conversion ratio from  $t'$  to  $t$  is then calculated using only the machines in  $M_c^{t, t'}$ . Specifically, we let  $A[t, c] = \sum_{m \in M_c^{t, t'}} y_m^t$  and  $A[t', c] = \sum_{m \in M_c^{t, t'}} y_m^{t'}$  for calculating  $C_{t' \rightarrow t}$ .

**Results.** To quantify the waste in resources, we compare the total capacity of the VMs that are being converted with the total capacity of the AV counts that are deducted by this conversion. In particular, let  $U_{original} = \sum_{t'} x_c^{t'} \cdot q_{t'}$  denote the total capacity of the VM types before conversion (i.e., “real” capacity of buffers), and  $U_{converted} = q_t \cdot \sum_{t'} x_c^{t' \rightarrow t}$  denote the converted capacity of these VMs.

**Theorem 1** *The conversion guarantees at least  $\frac{1}{4}$ -utilization of the converted AV counts’ capacity; explicitly:  $U_{original} \geq \frac{1}{4} \cdot U_{converted}$ .*

To prove Theorem 1, we will need the following auxiliary lemma that relates the conversion ratio with the sizes of the VM types.

**Lemma 1** *Under the above assumptions,  $C_{t' \rightarrow t} \leq 2 \cdot \frac{q_{t'}}{q_t}$ .*

**Proof of Lemma 1.** Consider a machine  $m$  that can currently fit at least one VM of type  $t$  or one VM of type  $t'$ . Let  $Q_m$  denote the available capacity of the bottleneck resource (e.g., CPU) of machine  $m$ . By definition,  $y_m^t = \lfloor \frac{Q_m}{q_t} \rfloor$  and  $y_m^{t'} = \lfloor \frac{Q_m}{q_{t'}} \rfloor$ . Then, the following are true:

$$q_t \cdot y_m^t \leq Q_m \quad \text{and} \quad q_{t'} \cdot (y_m^{t'} + 1) > Q_m.$$

Combining the above two inequalities, we obtain:

$$q_t \cdot y_m^t < q_{t'} \cdot (y_m^{t'} + 1) \Rightarrow \frac{y_m^t}{y_m^{t'}} < \frac{q_{t'}}{q_t} + \frac{q_{t'}}{q_t \cdot y_m^{t'}} \leq 2 \cdot \frac{q_{t'}}{q_t}$$

where the last inequality follows from the fact  $y_m^{t'} \geq 1$ .

Since this is true for all machines  $m$  that fit both VM types, we obtain:

$$A[t, c] = \sum_{m \in M_c^{t, t'}} y_m^t \leq 2 \cdot \frac{q_{t'}}{q_t} \sum_{m \in M_c^{t, t'}} y_m^{t'} = 2 \cdot \frac{q_{t'}}{q_t} \cdot A[t', c].$$

We can then easily see that:

$$C_{t' \rightarrow t} = \frac{A[t, c]}{A[t', c]} \leq 2 \cdot \frac{q_{t'}}{q_t}.$$

**Proof of Theorem 1.** We need to consider two cases.

- **Case 1:**  $C_{t' \rightarrow t} < 1$ . In this case, we obtain the following:

$$x_c^{t' \rightarrow t} = C_{t' \rightarrow t} \cdot x_c^t \leq 2 \cdot \frac{q_{t'}}{q_t} \cdot x_c^t.$$

We then get for the original and converted capacity of VM type  $t'$ :

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{2} \cdot q_t \cdot x_c^{t' \rightarrow t}.$$

- **Case 2:**  $C_{t' \rightarrow t} \geq 1$ . Similarly, in this case:

$$\begin{aligned} x_c^{t' \rightarrow t} &= \lceil C_{t' \rightarrow t} \cdot x_c^t \rceil \leq C_{t' \rightarrow t} \cdot x_c^t + 1 \leq C_{t' \rightarrow t} \cdot (x_c^t + 1) \\ &\leq 2 \cdot C_{t' \rightarrow t} \cdot x_c^t \leq 4 \cdot \frac{q_{t'}}{q_t} \cdot x_c^t. \end{aligned}$$

In the above, the second inequality follows from  $C_{t' \rightarrow t} \geq 1$ . The third inequality is due to  $x_c^t \geq 1$  (otherwise, no buffer of type  $t'$  would exist, clearly leading to zero waste in the conversion by default). Finally, the last inequality uses Lemma 1. As a result,

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot q_t \cdot x_c^{t' \rightarrow t}.$$

Putting both cases together, we can see that for all converted VM types  $t'$ , we have:

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot q_t \cdot x_c^{t' \rightarrow t}.$$


Summing over all such VM types:

$$\sum_{t'} q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot \sum_{t'} q_t \cdot x_c^{t' \rightarrow t} \Rightarrow U_{original} \geq \frac{1}{4} \cdot U_{converted}.$$

which concludes the proof of Theorem 1.



# Security and Performance in the Delegated User-level Virtualization

Jiahao Chen<sup>1,2\*</sup>, Dingji Li<sup>1,2,3\*</sup>, Zeyu Mi<sup>1,2</sup>, Yuxuan Liu<sup>1,2</sup>,  
Binyu Zang<sup>1,2</sup>, Haibing Guan<sup>4</sup>, and Haibo Chen<sup>1,2</sup>

<sup>1</sup>*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*

<sup>2</sup>*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

<sup>3</sup>*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*

<sup>4</sup>*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*

## Abstract

Today’s mainstream virtualization systems are plagued by severe security threats due to the large attack surface exposed by in-kernel hypervisor components such as KVM. To address this issue, this paper proposes a novel design called delegated virtualization, which decouples the commodity hypervisor into two planes: the *hypervisor plane* for hypervisor control (which is typically small and has fixed logic) and the *VM plane* for handling virtual machine (VM) requests and exceptions at runtime. As our investigation reveals that all known hypervisor vulnerabilities that threaten the host kernel lie in the VM plane, delegated virtualization completely offloads the in-kernel VM plane to a user-space hypervisor called DuVisor that directly interacts with its VM without exiting to the host kernel, based on a small hardware extension (481 lines of Chisel). We have implemented the hardware extension on an open-source RISC-V CPU on FireSim and built a Rust-based DuVisor atop it. The evaluation results demonstrate that DuVisor significantly reduces the attack surface with negligible performance overhead (< 5%). DuVisor’s source code is publicly available at <https://github.com/IPADS-DuVisor>.

## 1 Introduction

The technique of system virtualization, also known as virtualization, is essential for efficiently running concurrent virtual machines (VMs). Since its conception, virtualization has undergone three rough stages of evolution, gradually moving hypervisor functions outside of kernel mode. In the first stage (Figure 1-a), all hypervisor functions were implemented in kernel mode to multiplex scarce resources of large mainframe machines, such as the IBM VM/370 [44, 48, 54, 83]. In the second stage (Figure 1-b), mainstream hypervisors began to offload hypervisor functions to user mode, which issued system calls to take advantage of a host operating system (OS) [42, 43] or a management VM [40]. However, some functions still remained in kernel mode, such as instruction


emulation and memory virtualization. The third stage (Figure 1-c) began with the release of hardware extensions (e.g., Intel VMX [19] and AMD SVM [2]), which further reduced the kernel involvement in virtualization by shifting some virtualization functions to hardware.

Nowadays, third-stage hypervisors that are based on hardware extensions typically utilize a split model consisting of two cooperative components: a kernel-mode module and a user-mode helper. For instance, the most popular Linux/KVM-based virtualization system<sup>1</sup> includes a global KVM kernel module [49, 61] and a per-VM user-mode helper, such as QEMU [26]. The KVM module interacts with hardware extensions and the host kernel, while the user-mode helper is responsible for VM management and I/O virtualization.

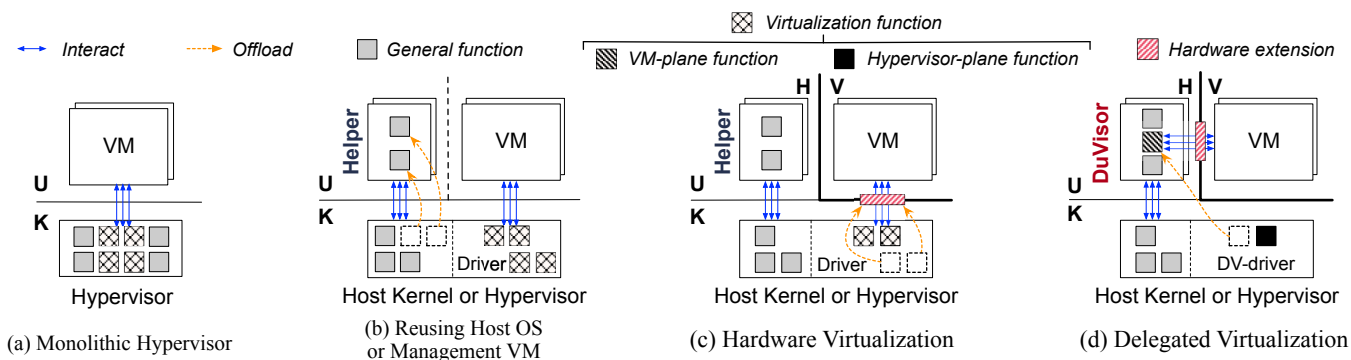
Unfortunately, vulnerabilities in the kernel-mode component of virtualization systems are discovered from time to time, making them a major threat to host security. For example, there have been more than a hundred CVEs reported in KVM during the course of its evolution [22]. These vulnerabilities can be exploited by a malicious VM to compromise the KVM component that interacts with the VM directly. The majority of these CVEs can be utilized to launch denial of service (DoS) attacks, causing host crashes and undermining the reliability of the host kernel as well as all co-located VMs [10, 16]. Even more concerning, once the KVM kernel module is hacked, the attacker may gain control of the entire system and carry out more severe attacks [3, 13]. In contrast, a compromised QEMU has a considerably smaller impact, usually limited to the current user-mode process and not affecting the host kernel or other VMs, thanks to the isolation between applications and kernel [24, 37].

The key idea of this paper is to **move all hypervisor components that directly interact with VMs at runtime to user space**, with the aim of minimizing the impact of any security bugs and reliability issues found in the user-mode hypervisor. However, there are three significant challenges

\* Co-first authors.

 Corresponding author: Zeyu Mi ([yzmizeyu@sjtu.edu.cn](mailto:yzmizeyu@sjtu.edu.cn)).

<sup>1</sup>Xen [40] and VMware products [36, 88] also exhibit a similar architecture.



**Figure 1: The architectural evolution of mainstream hypervisors (a → b → c) that gradually demote hypervisor functionalities out of kernel mode and the delegated virtualization proposed by this paper (d).** *U* and *K* represent user mode and kernel mode; *H* and *V* are hypervisor mode and virtualization mode introduced by hardware virtualization. The *Virtualization functions* includes ① *VM-plane functions* serving VMs directly at runtime, and ② *hypervisor-plane functions* performing resource control and error handling for the VM plane. The *General functions* comprises other VM-required functions such as virtualizing I/O devices. (a) Stage-1: The monolithic hypervisor puts all functions in kernel mode; (b) Stage-2: Offloading some functions to a user-mode helper process (e.g., I/O backend drivers), which can reuse host OS or management VM that manage hardware resources; (c) Stage-3: Offloading some virtualization functions into hardware (e.g., shadow paging → nested paging); (d) We propose the DuVisor approach that delegates all VM-plane functions (e.g., VM exit handling) to user space and minimizes direct interactions between the host kernel and VMs at runtime.

that make the design more complex than it initially appears: 1) *Privilege Restriction*: modern hardware virtualization extensions are only configurable in kernel mode, such as setting a VM’s stage-2 page table. This necessitates the presence of a hypervisor component in the host kernel to use these extensions. 2) *Security Risk*: simply moving all the management of VMs’ hardware resources to user mode violates the least privilege principle and enlarges the attack surface. For instance, if QEMU is allowed to modify a VM’s stage-2 page table, it can then access any physical memory pages, posing a significant security risk. 3) *Performance Overhead*: most VM exits are now forwarded by the kernel to the user-mode functions to handle. Then, the control flow must return to the kernel again to resume VMs’ execution, resulting in excessive runtime ring crossings and unacceptable performance costs [56, 91].

We identify that the tight coupling between hardware virtualization extensions and kernel mode is the root cause of the challenges mentioned above. Fortunately, we observe that recent hardware advancements have made it possible to expose many hardware resources that were previously only accessible by the kernel to user mode. One prominent example is that Intel has released user-level interrupts, which allow a user-level process to handle physical interrupts [20]. Another example is physical memory checking, such as RISC-V Physical Memory Protection (PMP) [32, 63], which limits the physical memory range a program can access. With these recent hardware trends, we believe it is time to retrofit existing hardware virtualization extensions and expose virtualization interfaces to user mode securely and efficiently, addressing the challenges mentioned above.

This paper proposes a hypervisor design principle of **decoupling the VM plane from the hypervisor plane**. The *VM plane* frequently interacts with VMs at runtime by han-

dling their VM exits, and enables various virtual resources through instruction emulation, nested paging, device virtualization, etc. In contrast, the *hypervisor plane* serves the VM plane with physical resource control (e.g., invoking kernel interfaces to manage resources) and fatal error handling. Following this principle, we propose a delegated virtualization architecture, which eliminates the notion that the kernel mode should provide VM abstractions. Instead, delegated virtualization offloads the VM plane that interacts directly with VMs into per-VM hypervisors running in user mode, called *DuVisor*<sup>2</sup>, while only leaving a tiny *DV-driver* in kernel mode responsible for the hypervisor plane.

As shown in Figure 1-d, we introduce a novel hardware extension called *Delegated Virtualization Extension (DV-Ext)* by slightly extending the existing hardware virtualization mechanism to securely expose hardware virtualization interfaces to user mode. Based on DV-Ext, all VM-plane functions in the existing hypervisor are offloaded to the user-mode *DuVisor* process. Specifically, *DuVisor* can directly utilize DV-Ext’s registers and instructions to serve runtime VM exits without trapping into the host kernel. On the other hand, the tiny *DV-driver* remaining in the kernel only wakes up occasionally to allocate physical resources or handle fatal errors for *DuVisor* processes.

*DuVisor* efficiently provides different virtualization functions in user mode with strong security guarantees. For CPU virtualization (§5.1), the *DuVisor* process creates a dedicated thread (*vthread*) for each virtual CPU (vCPU), and the *vthread* utilizes DV-Ext to handle this vCPU’s VM exits in user mode. For memory virtualization (§5.2), *DuVisor* configures a stage-2 page table for its VM and processes stage-2 page faults in user mode with a pre-allocated range of phys-

<sup>2</sup>Short for Delegated user-level HyperVisor

ical memory. The range used by a DuVisor and its VM is restricted by the DV-driver and DV-Ext via hardware physical memory checking. For I/O virtualization (§5.3), paravirtualized (PV) backend drivers in DuVisor directly communicate with their frontends in VMs. DuVisor further uses user-level posted interrupt to completely bypass the host kernel when sending notifications to its VM.

We have implemented DV-Ext based on a RISC-V Rocket CPU using FPGA. DV-Ext can be easily implemented by reusing existing hardware features, including hypervisor extension (H-Ext [28]) and user-level interrupts extension (N-Ext [33]). It only adds 481 lines of Chisel code. Based on DV-Ext, we use Rust to build DuVisor, and the code size is about 7K LoC. We also extend the Linux kernel v5.10.26 with a tiny DV-driver to cooperate with DuVisor by adding 337 LoC. Through software-hardware co-design, the kernel attack surface exposed to guest VMs is minimized, and any vulnerabilities that threaten existing hypervisors are now confined in the DuVisor process. Performance evaluation on cycle-accurate FireSim [59] shows that DuVisor incurs only negligible performance overhead for architectural operations and real-world applications.

In summary, the contributions of the paper are:

- We propose a delegated virtualization architecture that offloads the entire VM plane to the user-level DuVisor and leaves only a tiny DV-driver in the host kernel, minimizing the attack surface exposed to guest VMs and protecting the entire system from being impacted by the security and reliability issues in traditional hypervisors.
- We design a lightweight hardware DV-Ext that enables DuVisor to serve VM entirely in user mode.
- We implement the hardware extension on RISC-V with minimal modification and build a Rust-based DuVisor prototype.
- We evaluate the performance of DuVisor on AWS F1 FPGAs using cycle-accurate FireSim with a suite of real-world applications.

## 2 Background and Motivation

### 2.1 Hardware-assisted Virtualization

Mainstream hardware virtualization extensions [2, 4, 19, 28] offer comparable functionalities. We use RISC-V’s H-Ext as an exemplar due to its open-sourced implementations. As illustrated in Figure 1-c, H-Ext introduces two distinct modes, namely H mode and V mode, which are orthogonal to the existing privilege levels (U and K for user and kernel, respectively<sup>3</sup>). H mode is exclusively reserved for the hypervisor, while VMs operate in V mode. Only the hypervisor kernel mode (HK mode) is authorized to employ the virtualization interface for initiating, configuring, and resuming a VM, receiving traps from a VM to the hypervisor (also

<sup>3</sup>Although RISC-V kernel mode is referred to as “supervisor” (S) mode, we shall use the term kernel mode in this paper.

**Table 1: CVE analyses of KVM [22] and Xen [38].** *Host* stands for the vulnerabilities that could lead to an attack on the host kernel, including *PE* (privilege escalation), *DoS* (denial of service), and *DL* (data leakage). *Other* refers to CVEs that only attack guest VMs or cannot be exploited. *LoC* shows the code size in LoCs of each hypervisor. *Time Scale* indicates the year strides for each of the two hypervisor CVE analyses.

| Name | Total | Host |    |     | Other | LoC  | Time Scale |
|------|-------|------|----|-----|-------|------|------------|
|      |       | PE   | DL | DoS |       |      |            |
| KVM  | 111   | 21   | 7  | 52  | 31    | 142K | 2008-2022  |
| Xen  | 370   | 101  | 19 | 179 | 71    | 345K | 2007-2022  |

known as VM exits), injecting virtual interrupts into a VM, and installing a stage-2 page table (S2PT). To control VMs, the helper process in hypervisor user mode (HU mode) must issue system calls to invoke functions provided by the kernel driver (e.g., KVM) in HK mode.

In this paper, we define the VM plane as all VM-serving functions that handle VM exits and virtualize resources at runtime, while referring to the hypervisor plane as the set of all hypervisor-serving functions that initialize the VM plane, manage physical resources, and handle emergency events. The VM plane in existing hypervisors spans the HK mode (e.g., CPU and memory virtualization) and the HU mode (e.g., device virtualization), whereas the hypervisor plane is located solely in the HK mode. Whenever a VM exit occurs, the hardware first switches the CPU control flow to the in-kernel VM plane. Most VM exits can be handled directly in the HK mode without switching to the user-mode helper. For instance, in the case of a stage-2 page fault (#S2PF), the hypervisor plane obtains a physical page, and the in-kernel VM plane inserts a new address mapping to the VM’s stage-2 page table before resuming the VM directly. However, other VM exits, such as some Memory-Mapped I/O (MMIO) trapings, cannot be entirely resolved by the in-kernel VM plane and must be forwarded to the user-level helper for emulation.

### 2.2 Vulnerabilities of Hypervisors

In contrast to the user-mode helper, which features a clear isolation boundary with the kernel, vulnerabilities arising from the in-kernel components of hypervisors pose significantly more severe threats to the host kernel. This is due to the fact that these components possess the highest level of privilege and interact directly and most frequently with VMs during runtime, thereby exposing a greater number of attack surfaces to VMs. The in-kernel components of hypervisors have accumulated a considerable number of publicly revealed vulnerabilities, underscoring their weak security and fault isolation. While there are also many vulnerabilities in the non-hypervisor components of the host kernel, this paper primarily focuses on hypervisor vulnerabilities, with non-hypervisor vulnerabilities being discussed in §7 and §9. Table 1 presents the statistics of disclosed vulnerabilities in KVM [22] and Xen [38], highlighting three key characteris-



**Table 2: CVE classification based on subsystems of KVM.** The data excludes 31 CVEs in the *Other* column of Table 1 that do not harm the host kernel. *Original* represents the original number of host-attacking CVEs in the KVM. *After Offload* means the number of host-attacking CVEs that remain in HK mode after offloading most components to HU mode by existing works [87, 91].

|                  | Subsystem                 | Number of CVEs |               |
|------------------|---------------------------|----------------|---------------|
|                  |                           | Original       | After Offload |
| VM Plane         | Memory Virtualization     | 10             | 2             |
|                  | Interrupt Virtualization  | 18             | 18            |
|                  | ISA Emulation             | 19             | 10            |
|                  | Para-Virtualization       | 4              | 0             |
|                  | VM Exit Handling          | 17             | 17            |
|                  | Device Virtualization     | 12             | 0             |
| Hypervisor Plane | Hypervisor Initialization | 0              | -             |
|                  | Resource Control          | 0              | -             |
|                  | Emergency Handling        | 0              | -             |

tics of the vulnerabilities in the in-kernel components.

- **Large Vulnerability Quantity.** There are 111 and 370 disclosed CVEs in KVM and Xen, respectively. 72.07% and 80.81% of them cause kernel-level exploits, including information leakage, host DoS [10, 16, 84], and privilege escalation of a guest VM [3, 12, 13, 27, 82, 86].
- **Severe Security Threats.** Due to the high privilege of in-kernel components, their vulnerabilities can easily crash the entire system, disrupting the execution of other co-located VMs. As shown in Table 1, 65.77% and 75.68% of CVEs (i.e., PE and DoS) in KVM and Xen, respectively, can be exploited to launch DoS attacks via vulnerabilities such as NULL pointer dereferences [17] or out-of-bounds reads/writes [15]. Even worse, carefully crafted exploits may enable attackers to escalate a VM’s privilege and compromise the entire system, including all other VMs.
- **Low Exploit Costs.** CVE exploits can be researched and crafted at relatively low costs by well-financed experts who are motivated by the significant profit potential of successful attacks. For instance, it took just two months for a Google expert to develop an exploit that enabled VM escape via a vulnerability in KVM code [3].

### 2.3 Limitations of Deprivileged Execution

A long line of work has attempted to deprivilege the in-kernel functionalities of hypervisors to mitigate the threats of vulnerabilities in existing hypervisors [46, 80, 87, 91]. For example, NOVA [87] builds a microhypervisor based on the microkernel architecture. DeHype [91] endeavors to demote most parts of KVM into user mode while leaving a HypLet in kernel mode because the sensitive hardware virtualization instructions can only be executed in this mode. However, such deprivileged execution methods have two limitations:

**Non-eliminable In-kernel Vulnerabilities.** We investigated the 80 host-attacking CVEs of KVM from Table 1 and identified the subsystems in which they are present. As

**Table 3: Breakdown of the latency of handling an MMIO read in QEMU/KVM on ARM, RISC-V and x86-64.** *Kernel* represents the cycles spent on the in-kernel transfer operations. *User* stands for the cycles consumed by the I/O emulation and VM entry/exit.

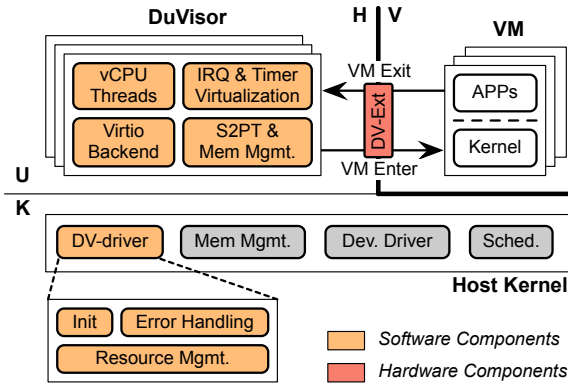
| Platform | Kernel | User  | Total |
|----------|--------|-------|-------|
| ARM      | 4,323  | 1,596 | 5,919 |
| RISC-V   | 3,135  | 4,067 | 7,202 |
| x86-64   | 2,415  | 1,704 | 4,119 |

shown in the *Original* column of Table 2, these CVEs are distributed throughout all VM-plane subsystems, whereas none of them exist in the hypervisor plane. We further examined whether these CVEs could be addressed by prior works [87, 91] that attempted to offload as many VM-plane components as possible to user space. The CVE number that remains in the host kernel after offloading is displayed in the right-most column of Table 2. Unfortunately, the majority (58.75%) of CVEs cannot be eliminated because the VM-plane subsystems in which they reside must operate in the HK mode due to hardware privilege restrictions. For example, the interrupt virtualization subsystem accesses privileged registers, so it must remain in the HK mode. Similarly, some memory virtualization functions, particularly those configuring sensitive stage-2 page tables for VMs, must also remain in the HK mode. As a result, the in-kernel VM plane poses entrenched security and reliability risks to the host kernel.

**Redundant and Costly Mode Switching.** Because the hypervisor must use the kernel component to drive the hardware virtualization extension, moving most of the kernel component to user space will result in more frequent and expensive interactions between the VM and the user-level hypervisor due to the kernel’s involvement, leading to higher performance overhead. To understand the cost associated with kernel involvement, we break down the handling procedure of an MMIO read operation in QEMU/KVM to illustrate the VM-VMM communication cost and find that 73.04%, 43.53%, and 58.63% of CPU cycles are consumed by in-kernel transfer operations on ARM, RISC-V, and x86-64, respectively (Table 3). As a result, minimizing the host kernel part by delegating more kernel functions to user space [87, 91] will lead to significant performance overhead due to the expensive VM-VMM communication costs on each VM exit handling.

## 3 System Design Overview

In this paper, we introduce **delegated virtualization** to safeguard the overall security and reliability of virtualization systems by preventing compromised hypervisor components from directly breaching the host kernel. To circumvent the privilege restrictions of existing hardware virtualization, delegated virtualization simply exposes the existing virtualization interfaces to user mode without requiring intrusive hardware modifications. We explicitly decouple the VM plane from the hypervisor plane and offload all VM-plane func-



**Figure 2: The architecture overview of DuVisor.** *DV-Ext* is Delegated Virtualization Extension.

tions to the user-mode DuVisor. A tiny DV-driver in the host kernel serves as the hypervisor plane, which is removed from all runtime interactions (VM plane) between DuVisor and its VM.

The design of delegated virtualization offers two significant advantages. First, it minimizes the attack surface of the host kernel’s hypervisor components accessible to VMs and confines hypervisor vulnerabilities to user space, largely improving security and fault isolation between the VMs and the host kernel. Furthermore, the security and reliability benefits are obtained without any performance penalty due to the direct runtime interactions between a guest and its hypervisor.

The architecture of delegated virtualization, as depicted in Figure 2, comprises three primary components: the Delegated Virtualization Extension (DV-Ext), per-VM DuVisor hypervisor processes, and a global DV-driver in the kernel.

The Delegated Virtualization Extension (DV-Ext) must be installed on the hardware (§4). It empowers the host kernel to determine whether or not to delegate hardware virtualization functions to HU mode. If the delegated mode is enabled, the hardware virtualization interface can be accessed by unprivileged software without trapping into the host kernel. If it is not enabled, DV-Ext functions similarly to traditional hardware virtualization for compatibility.

An HU-mode DuVisor process leverages the hardware interface exposed by DV-Ext to control an unmodified VM (§5). To support the normal execution of a VM, DuVisor dynamically virtualizes physical resources to handle runtime VM exits. In this paper, this workflow is defined as the VM plane and is handled by the DuVisor process in HU mode. Moreover, to support memory virtualization in HU mode, the DuVisor process builds a stage-2 page table for its VM. If stage-2 page faults occur due to missing or illegal page mappings, DuVisor dynamically adds or updates mapping entries in the stage-2 page table. Like conventional hypervisors, DuVisor spawns distinct user-level threads for each vCPU, referred to as *vthreads*. It also supports PV I/O devices and virtual interrupts (including timers) for this VM. DuVisor can depend not only on the host kernel to manage external de-

**Table 4: The registers and instructions added (or modified) by DV-Ext.** The lowercase and uppercase names stand for registers and instructions. The registers starting with “hu” are accessible in HU mode while those starting with “h” can only be accessed in HK mode. The two instructions can be invoked in HU mode.

| Type         | Mode    | Name                                    | Description                             |
|--------------|---------|-----------------------------------------|-----------------------------------------|
| Registers    | HU      | hu_er                                   | VM exit reason                          |
|              |         | hu_einfo                                | Additional information about a VM exit  |
|              |         | hu_vpc                                  | IP address of a faulted vCPU            |
|              |         | hu_ehb                                  | Base address of the VM exit handler     |
|              |         | hu_vcpuid                               | The vCPU ID running on a physical core  |
|              | hu_vitr | Virtual interrupt number to be inserted |                                         |
| Instructions | HU      | HURET                                   | Resume the vCPU execution               |
|              |         | HUFLUSHGPA                              | Flush TLB entries associated with a GPA |
|              |         | h_enable                                | Turn on DV-Ext                          |
| HK           | HK      | h_deleg                                 | Delegate VM exits to HU mode            |
|              |         | h_vmids                                 | The VM ID running on a physical core    |

vices like storage media and network cards but also control devices in HU mode by DPDK [18] to boost I/O virtualization.

A tiny DV-driver is inserted into the host kernel (§6) as the hypervisor plane, which occasionally participates in the management of physical resources for each DuVisor without interfering with runtime VM exits processing. Specifically, the DV-driver uses DV-Ext to enable/disable delegated mode and allocates resources (such as physical memory) for DuVisor processes. To mitigate security risks, it also restricts DuVisor’s physical memory view and handles emergencies, such as exceptions triggered by illegal physical memory accesses by untrusted VMs. DuVisor still relies on the host kernel to schedule all its threads and VMs.

**Assumptions and Threat Model.** We assume that the hardware (including DV-Ext) is correctly implemented and trusted. The goal of DuVisor is to defend the host kernel against malicious VMs, so that the host kernel and the DV-driver are trusted as well. However, in a multi-tenant cloud environment, a guest VM controlled by a hostile tenant may exploit vulnerabilities in DuVisor to compromise the hypervisor. Therefore, the user-level DuVisor process is considered untrusted by the host kernel. Side-channel attacks [73, 74] and corresponding defense methods [37, 45, 76] are orthogonal to the design of DuVisor and are not considered in this paper.

## 4 Delegated Virtualization Extension

In this section, we describe the design of DV-Ext, which lifts the restriction that hardware virtualization extensions are inaccessible to user mode. DV-Ext provides hardware interfaces to user-level DuVisor for obtaining VM information and controlling VM behaviors. These hardware interfaces can take different forms on varied hardware architectures. This section elaborates on register-based hardware interfaces as an example to present a detailed design of DV-Ext, which is suitable for the RISC-V and ARM architectures mentioned earlier. Additionally, we discuss the design of hardware interfaces based on memory and specialized instructions in §9 to

demonstrate DV-Ext’s universality. Table 4 shows the registers and instructions of DV-Ext.

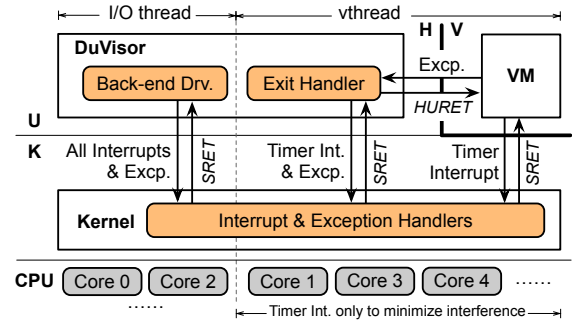
**HU-mode Registers and Instructions.** We observe that certain privileged registers are only configured during hypervisor initialization and are rarely accessed at runtime. Therefore, these registers can be considered hypervisor-plane registers and are not exposed to HU mode. The remaining registers, however, are frequently used during VM runtime and are defined as VM-plane registers. Accordingly, DV-Ext allows HU mode to access these registers without restriction.

VM-plane registers, as shown in Table 4, are denoted by names beginning with “*hu*”. They are accessible in HU mode when HK mode activates DV-Ext by setting up the *h\_enable* register. The VM-plane registers are classified into two categories. The first category records VM information for VM exits, such as *hu\_er* and *hu\_einfo*, which the hypervisor reads for handling VM exits. The second category controls the runtime behaviors of the hypervisor or VMs. For example, a hypervisor can configure *hu\_vitr* to inject a virtual interrupt to a vCPU.

**Delegatable VM Exits.** DV-Ext provides delegatable VM exits (DVE), which enables a VM to immediately trap to its DuVisor process in HU mode. The kernel mode can configure DVEs by modifying the *h\_deleg* register, whose individual bits regulate the delegation of specific types of VM exits. For instance, the DV-driver in HK mode can delegate stage-2 page faults and sensitive instruction faults such as WFI (i.e., wait-for-interrupt instruction for entering low-power standby CPU state, similar to HLT on x86) to HU mode by setting the corresponding bits in *h\_deleg*. When a DVE occurs, the hardware searches for a hypervisor handler using the address specified in *hu\_ehb*. DV-Ext additionally provides the *HURET* instruction for HU mode to resume VM execution after handling a DVE, and the entry point of the VM is stored in the *hu\_vpc* register.

**HU-mode Memory Virtualization.** Since the register storing the base address of a stage-2 page table is rarely modified after a VM is booted, DV-Ext does not expose it to HU mode. However, in HU mode, the user-level hypervisor can still freely update stage-2 translation by exposing the in-memory stage-2 page table to HU mode. Therefore, the page table format is consistent with the original stage-2 page table used in HK mode. As HU mode needs to flush TLB entries after updating stage-2 translation, DV-Ext exposes a TLB maintenance instruction to HU mode as the *HUFLUSHGPA* instruction, which can flush TLB entries associated with a specific GPA and VMID.

**Exitless Interrupt Virtualization.** Posted interrupt allows the hypervisor to deliver virtual interrupts to a running vCPU without VM exit. DV-Ext enables user-level posted interrupt that DuVisor can directly utilize for injecting virtual external interrupts in HU mode. DuVisor should specify the receiving vCPU’s VCPUID and the interrupt vector in *hu\_vitr*. Similarly, DV-Ext supports V-mode posted interrupt that a



**Figure 3: Exception and physical interrupt handling when running a DuVisor VM.** I/O threads and vthreads can be dynamically scheduled on different CPU cores. To minimize the involvement of the host kernel as well as the interference of physical interrupts, the DV-driver can configure those cores that run vthreads (e.g., core 1/3/4) to trigger solely timer interrupts. For other physical interrupts such as external interrupts generated by I/O devices, it is natural to route them to cores that run I/O threads of the backend drivers (e.g., core 0/2).

vCPU can issue a virtual inter-processor interrupt (IPI) without VM exit by configuring *hu\_vitr*. To prevent misdelivery, the hardware checks the VMID in *h\_vmids* and the VCPUID information pre-configured by the DV-driver before triggering a virtual interrupt. An illegal operand triggers a fault into HK mode to wake up the DV-driver. DV-Ext also adds virtual timer interrupts that can be triggered without VM exit.

## 5 DuVisor Design

### 5.1 Handling VM Exits

VM exits are caused by either exceptions or physical interrupts. In existing virtualization systems, all VM exits are trapped to the in-kernel hypervisor component for handling, but this is not the case in DuVisor. In contrast, all exceptions that result in VM exits are sent to the user-level DuVisor, while physical interrupts continue to be trapped and handled by the host kernel. It is inappropriate to direct physical interrupts to HU mode (DuVisor) due to their vital importance to the host kernel for tasks such as scheduling and device management, as HU mode is not trusted.

Figure 3 illustrates how exceptions and physical interrupts are handled when running a DuVisor VM. During the preparation of the VM execution environment, I/O threads of the backend driver and vthreads are scheduled to different CPU cores. After preparation, vthreads in DuVisor execute an *HURET* instruction to enter V mode and start running guest code until a VM exit occurs. For VM exits caused by synchronous exceptions, the CPU control flow directly traps from the VM to DuVisor’s VM exit handler. The handler then determines the exception type by accessing corresponding HU mode registers provided by DV-Ext. After handling the VM exit, the vthread resumes the VM by executing *HURET* again.

On the other hand, VM exits caused by physical interrupts

are directed to the interrupt handler in HK mode. To ensure that the register states of DuVisor VMs are not corrupted due to scheduling, the DV-driver saves all DV-Ext-related registers before handling the physical interrupt and restores them before directly returning to V mode with an *SRET* instruction. Due to the small number of DV-Ext-related registers, the performance impact from this additional save/restore logic is minimal (§8.4).

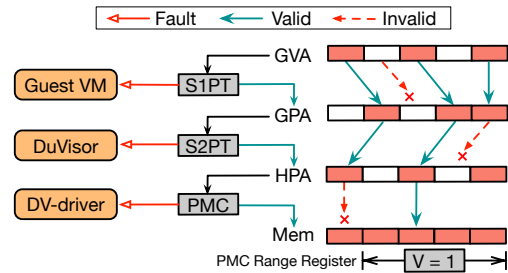
To minimize interference, we configure the DV-driver so that only physical timer interrupts periodically occur on cores where vthreads are running. For physical external interrupts generated by physical devices, a PV I/O device in DuVisor relies on the in-kernel device driver to handle them. Hence, it is natural to direct these external interrupts to the same cores as the I/O threads of the backend driver. Moreover, if IOMMU [29] is available, a physical device can be passthrough into the VM for better performance, which sends posted interrupts that directly inject physical external interrupts to the VM without VM exit. Additionally, there are few physical IPIs between vthreads and I/O threads, thanks to the exitless interrupt virtualization (§5.3). Therefore, only physical timer interrupts may periodically trigger on cores running vthreads and bring the running VM into HK mode.

## 5.2 Restricted Memory Virtualization

In contrast to traditional hypervisors' in-kernel stage-2 page table management, a DuVisor process handles stage-2 page faults and provides memory virtualization in HU mode without involving the kernel. To establish mappings of guest physical addresses (GPAs) for the VM, DuVisor populates stage-2 page table entries with host physical addresses (HPAs) in HU mode. Therefore, it requests the DV-driver to allocate contiguous memory regions with HPA information. Each stage-2 page fault traps to DuVisor, which then adds a free physical page from the pre-allocated memory region into the VM's GPA space by updating its stage-2 page table.

One natural challenge is how to prevent the untrusted DuVisor from maliciously configuring the stage-2 page table to access arbitrary HPA. A malicious DuVisor, for instance, may allow its VM to access (or alter) sensitive data in another VM's memory by directly mapping the attacker VM's GPA to the victim's HPA. Worse, the rogue VM can use this method to read and modify the host kernel memory. This issue can be addressed with a straightforward technique that requires DuVisor to manage a fake stage-2 page table instead of the real one. DuVisor only manages the fake stage-2 page table and must invoke system calls to ask the DV-driver to check this table and synchronize it to the real one. Although this method sounds reasonable, it frequently involves the kernel at runtime, leading to significant costs for memory-intensive workloads. Moreover, it complicates the memory management of the DV-driver.

We adopt an alternative approach, allowing DuVisor to manage the real stage-2 page table freely in HU mode



**Figure 4: The translation and checking procedure of a memory access request issued from the V mode.**

without entering kernel mode. Emerging hardware mechanisms, such as Intel TDX's Physical Address Metadata Table (PAMT) [47], AMD SEV-SNP's Reverse Mapping Table (RMP) [1], and ARM CCA's Granule Protection Table (GPT) [5], can dynamically restrict access to physical memory. Take RISC-V Physical Memory Protection (PMP) for example, it checks every memory access against up to 64 PMP entries configured on each core. Inspired by this, we propose to utilize such physical memory checking (PMC) mechanisms to limit the physical memory range that VMs can access.

With PMC, we can allow DuVisor to configure its stage-2 page table optimistically. The MMU automatically checks whether the target HPAs of the VM's memory accesses exceed the range limit of the pre-allocated physical memory regions. If so, the MMU triggers a fault to wake up the DV-driver. This design eliminates the stage-2 memory management module in the DV-driver. Furthermore, the overhead of dynamic checking is negligible since it is achieved by merely comparing offsets.

However, the current PMC technique is not specifically designed to restrict VM memory accesses. It examines every HPA access issued from the current physical core, which also restricts the host kernel and DuVisor from using physical addresses out of the configured entries. This is a significant constraint because the host OS can map the virtual addresses of the kernel and DuVisor to arbitrary physical memory addresses, which may exceed the ranges specified by the limited number of PMC regions. To overcome this constraint, DV-Ext extends the existing PMC mechanism slightly to make it work only for HPAs targeted by the V-mode memory accesses. DV-Ext adds a "Virtualization" (V) bit to each of the current PMC range registers. If the bit is set, the PMC only verifies the V-mode memory accesses according to the range registers.

Figure 4 illustrates how a guest virtual address (GVA) is translated into an HPA and finally reaches physical memory. The GVA is first translated into a GPA via MMU hardware according to the stage-1 page table (S1PT) built by the guest VM. Similarly, the GPA is translated into an HPA referring to the stage-2 page table controlled by DuVisor. A translation failure in the stage-2 page table triggers a stage-2 page fault into DuVisor for handling. Eventually, the PMC pre-



configured by the DV-driver checks the output HPA before accessing physical memory.

The DV-driver is responsible for allocating multiple physical memory regions for each VM, with each region being protected by a single PMP region. If the HPA exceeds the memory range specified by all PMP regions, the PMC will generate an exception and awaken the DV-driver to handle the situation. For example, the DV-driver may terminate the VM that triggered the exception and proceed to run other VMs. As the per-core PMP configurations are tied to a specific VM, the host kernel must save the PMP register values of the previous VM and install those of the next VM when switching between them.

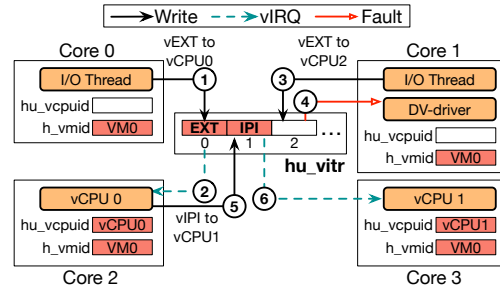
### 5.3 I/O and Interrupt Virtualization

I/O virtualization in DuVisor works similarly to existing approaches. It supports PV (e.g., virtio) and emulated (e.g., tty) I/O devices for its VM. Although DuVisor is compatible with passthrough devices, its current implementation does not support them due to the lack of IOMMU on the RISC-V platform. However, we can easily support them when IOMMU becomes available (§ 9).

For each PV and emulated device, DuVisor spawns dedicated I/O thread(s) during VM initialization. These threads are responsible for responding to VM I/O requests and interacting with host I/O devices. For instance, a TX thread is launched for a virtio network device’s TX queue to handle network packets from the guest VM. To reduce the kernel attack surface, DuVisor can be combined with kernel-bypass virtio backends such as vhost-user. In particular, the RX thread keeps polling the NIC in HU mode to receive incoming network packets and notifies the guest VM through virtual external interrupts (vEXTs).

To enable efficient PV I/O notifications, DV-Ext supports directly injecting vEXTs into a running VM through user-level posted interrupt. Posted interrupt is the most efficient mechanism for interrupt virtualization, which allows a virtual interrupt to be injected into a running VM without triggering VM exits. However, on existing hardware, a hypervisor must enter kernel mode to send a posted interrupt, which means that communications between vCPUs and between the HU-mode helper and its VM must go through the host kernel, contrary to the design principle of DuVisor. By contrast, the user-level posted interrupt in DV-Ext does not require kernel participation. Specifically, the I/O thread injects vEXTs by writing the interrupt vector to the posted interrupt register in HU mode. If the target vCPU is running on a core, DV-Ext immediately triggers the vEXT on that core. Otherwise, DV-Ext records the vEXT information and does not deliver it until the target vCPU resumes execution.

The DV-driver assigns a unique VMID to each DuVisor during initialization and writes this VMID to the per-core *h\_vmid* register every time a DuVisor is scheduled on a physical core. The VMID ensures that DuVisor’s I/O threads can



**Figure 5: Exitless interrupt virtualization with user-level posted interrupt and V-mode posted interrupt.** *vEXT* represents virtual external interrupt and *vIPI* represents virtual IPI.

only send posted interrupts to vCPUs with the same VMID as theirs. Since *h\_vmid* is only accessible in HK mode, the DuVisor process and the guest VM cannot modify its value. Each vCPU has its VCPUID, which the guest kernel writes to *hu\_vcpuid* during the boot process of each corresponding vCPU. The VCPUID is used by DV-Ext to identify the core on which the target vCPU is executing before delivering an interrupt.

Figure 5 depicts how a vEXT is delivered using user-level posted interrupts. For example, the I/O thread on core 0 attempts to insert a vEXT to vCPU 0 by writing the target vCPU’s associated location in *hu\_vitr* ①. DV-Ext then finds that vCPU 0 is executing on core 2 and generates a vEXT on core 2 ②. If the sending thread attempts to send a wrong VCPUID or has a different VMID from the receiver, writing to *hu\_vitr* will trigger a fault and wake up the DV-driver to handle this issue. For instance, the I/O thread on core 1 attempts to deliver a vEXT to a nonexistent vCPU 2 ③, and DV-Ext identifies this as an invalid operation and informs the DV-driver to handle the fault ④.

In addition to vEXTs, a multi-vCPU VM also requires efficient virtual IPIs (vIPIs) for inter-vCPU communications. To this end, DV-Ext supports V-mode posted interrupts that allow both sender and receiver vCPUs to incur no VM exit for a vIPI. Figure 5 also shows how a vIPI is generated using V-mode posted interrupts. The VMID and the VCPUID have the same effect as they do in user-level posted interrupt cases. Specifically, the vIPI issued from vCPU 0 on core 2 to vCPU 1 via writing *hu\_vitr* ⑤ is triggered by DV-Ext on core 3, where vCPU 1 is running ⑥.

Furthermore, virtual timer interrupts (vTimers) are necessary for each DuVisor VM. DV-Ext supports directly firing an expired vTimer inside the VM. Currently, a DuVisor VM can receive vTimers without VM exits, but it must trap to the DuVisor to set up timer events. Although the hardware can further remove the VM exits of setting timer events, we do not consider it necessary because the infrequent vTimers have little impact on VM performance.

## 6 Implementation

### 6.1 DV-Ext Implementation

We chose the RISC-V platform to implement DV-Ext because it has rich open-sourced implementations of system-on-chip (SoC). We used a 5-stage in-order scalar processor, specifically the RISC-V Rocket Core [34], with a configuration of 16KB L1 ICache, 16KB L1 DCache, 512KB shared L2 cache, and 16GB DRAM.

DV-Ext does not require intrusive modifications to the CPU hardware to implement these VM-plane registers and instructions. Specifically, *hu\_er*, *hu\_einfo*, *hu\_vpc*, and *hu\_ehb* are aliases of *ucause*, *utval*, *uepc*, and *utvec* from RISC-V N-Ext (i.e., the user-level interrupts extension), and *HURET* is implemented based on N-Ext’s *URET*. The *HU-FLUSHGPA* instruction is implemented by exposing H-Ext’s *HFENCE.GVMA* to the HU mode. Similarly, *h\_enable* and *h\_deleg* are implemented by extending H-Ext’s *hstatus*, *hideleg*, and *hedeleg*. Therefore, most architectural implementations for these registers and instructions can be reused. Only *hu\_vcpuid*, *hu\_vitr*, and *h\_vmid* are newly added registers for exitless interrupt virtualization.

Our DV-Ext implementation added 481 lines of Chisel to extend the existing H-Ext implementation [35] and support DVE. Additionally, we added 14 lines of Chisel to extend RISC-V PMP for VM memory restriction.

### 6.2 Software Implementation

Our prototype system of DuVisor consists of 7,128 LoC (5,052 lines of Rust, 166 lines of assembly, and 1,910 lines of C). The code of libraries we use is not included in the implementation effort. To implement the virtualization of CPU, memory and interrupt, 4,984 lines of Rust and 166 lines of assembly were written, in which the assembly is used to access architecture-dependent registers. For the virtualization of I/O devices, we ported the I/O backend of virtio block and virtio network devices from the *kvmtool* to DuVisor to reduce coding effort, accounting for 1,287 lines of C. We applied our design (e.g., the user-level posted interrupts) to these virtual devices as well as made some optimizations. Since there is no available DPDK support for RISC-V platforms currently, We also extended the virtio network backend with a user-space NIC driver using 623 lines of C to achieve a relatively fair performance comparison with KVM’s mature *vhost-net* backend. These I/O backend implementations comprise 1,910 lines of C in total.

We wrote a tiny Linux kernel module to work as the DV-driver and it has 337 LoC. DV-driver provides an *ioctl* system call for DuVisor to request several services. First, the DV-driver detects whether the hardware supports DV-Ext and enables it when a user process requests it. Second, it sets up the *h\_deleg* register to configure DVEs. Third, the DV-driver allocates contiguous physical memory regions for DuVisor from Linux’s contiguous memory allocator (CMA) and pins

**Table 5: CVEs in different KVM subsystems that can be successfully exploited in NOVA/DeHype and DuVisor’s architecture.** We omit 31 CVEs that cannot attack the host kernel.

| Subsystem                | KVM |           |    | NOVA/DeHype |           |    | DuVisor |          |    |
|--------------------------|-----|-----------|----|-------------|-----------|----|---------|----------|----|
|                          | PE  | DoS       | DL | PE          | DoS       | DL | PE      | DoS      | DL |
| Memory Virtualization    | 3   | 6         | 1  | 1           | 1         | 0  | 0       | 0        | 0  |
| Interrupt Virtualization | 3   | 13        | 2  | 3           | 13        | 2  | 0       | 0        | 0  |
| ISA Emulation            | 4   | 14        | 1  | 3           | 7         | 0  | 0       | 0        | 0  |
| Para-Virtualization      | 0   | 4         | 0  | 0           | 0         | 0  | 0       | 0        | 0  |
| VM Exit Handling         | 6   | 11        | 0  | 6           | 11        | 0  | 0       | 0        | 0  |
| Device Virtualization    | 5   | 4         | 3  | 0           | 0         | 0  | 0       | 0        | 0  |
| <b>Sum</b>               |     | <b>80</b> |    |             | <b>47</b> |    |         | <b>0</b> |    |

the physical memory regions to ensure their availability at runtime. Before returning to HU mode, the DV-driver also dives into the M-mode OpenSBI and configures the PMP entries to restrict the VM’s physical memory access range. Each PMP entry is set up with a *pmpcfg* register specifying the V bit and memory accessibility as well as a *pmpaddr* register recording the physical address and length. The host kernel should also have a PMP fault handler that terminates the fault process gracefully. Currently, our prototype does not implement such a fault handler for simplicity, but it is not hard to extend the existing exception handler to implement one. Lastly, the DV-driver initializes a VMID for each DuVisor process that will be used by interrupt virtualization.

Based on the Linux kernel which already switches the general purpose registers and V-mode CSRs, we further modified the context switch logic (74 LoC) to save and restore the DV-Ext registers if the process has enabled DV-Ext. If the next scheduled thread is not a vthread from the same VM, the PMP registers are also switched.

## 7 Security Analysis and Evaluation

In this section, we analyze the overall system security of DuVisor from the perspective of an attacker.

**Attack from Guest to Host Kernel.** A hostile VM can exploit vulnerabilities to compromise the hypervisor. If these vulnerabilities are exploited in kernel mode, the attacker can achieve VM escape, steal sensitive kernel data, and even crash the entire kernel. Table 5 shows such CVEs in different KVM subsystems and how many of them can be successfully exploited in NOVA [87]/DeHype [91] and DuVisor’s architecture. NOVA and DeHype, limited by hardware, cannot fully move these subsystems to user mode, thus still leaving 58.75% of the vulnerabilities undefended. In contrast, DuVisor has deprivileged all of these subsystems in HU mode, reducing the host kernel’s attack surface and preventing any of these CVEs from directly jeopardizing it.

Table 6 lists typical vulnerabilities that we evaluated on DuVisor. Their security and reliability threats are confined within the DuVisor processes. Therefore, they cannot harm the host kernel directly.

**Attack from Guest to DuVisor.** In theory, a malicious guest can exploit vulnerabilities to attack the user-level DuVisor. To enhance security, DuVisor is developed in Rust, a

**Table 6: Case studies of KVM CVEs.** This table lists 6 representative KVM CVEs that have the potential to disrupt the normal execution of the host kernel, resulting in DoS or even more severe attacks. To evaluate the impact of these CVEs on a system running DuVisor, we emulated the vulnerabilities in the corresponding subsystems of DuVisor. The results show that these CVEs only cause DuVisor itself to crash, while the host kernel can continue to execute other programs (including DuVisor VMs) correctly.

| CVE-*      | Attack Effect                                                                                                                                                                      |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2017-12188 | <b>Memory Virtualization:</b> KVM’s misconfiguration of the stage-2 page table allows the guest to access host memory, which can cause the host OS to crash or even be controlled. |
| 2018-16882 | <b>Interrupt Virtualization:</b> A use-after-free issue in the posted interrupt handling may allow hostile VMs to execute arbitrary code in HK mode.                               |
| 2016-8630  | <b>ISA Emulation:</b> Improper implementation for instruction decoding of KVM may cause host kernel crashes and jeopardize the availability of the host OS.                        |
| 2020-8834  | <b>VM Exit Handling:</b> KVM’s imperfect isolation of the guest states allows malicious VMs to corrupt the stack and destroy the availability of the entire system.                |
| 2016-5412  | <b>Para-Virtualization:</b> The incorrect implementation of a hypercall in KVM could lead to an infinite loop that crashes the host kernel.                                        |
| 2019-6974  | <b>Device Virtualization:</b> A use-after-free vulnerability in device virtualization may lead to VM escape in the kernel.                                                         |

high-performance language that guarantees memory safety and thread safety. This greatly reduces the security risks associated with memory vulnerabilities [6, 7, 9, 12, 14] and threading vulnerabilities [8, 11, 14], such as the use-after-free bugs [9, 12] in traditional hypervisors written in C/C++. In addition, a vulnerable DuVisor can be promptly patched in user space without rebooting the host OS.

**Attack from DuVisor to Host Kernel.** Although the DuVisor design prevents a malicious VM from directly compromising the host kernel through the in-kernel hypervisor component, the VM may still attempt to attack the host kernel after controlling the DuVisor. Various existing techniques can be leveraged to defend against such user-level attacks, which are orthogonal to the DuVisor design. The static resource allocation and vhost-user devices in DuVisor significantly reduce the system calls. For example, DuVisor only requires 17 system calls to serve a Linux VM at runtime. Therefore, the kernel can use seccomp [24] to effectively restrict the system calls and their parameters that a DuVisor can invoke at runtime. The host kernel can also be reconstructed as a microkernel to improve its isolation, although this is beyond the scope of this paper.

Furthermore, neither the DV-driver nor the DV-Ext interface gives DuVisor additional capabilities to compromise the host kernel. First, the DV-driver has a small enough code base that it could be formally verified. Second, although the DV-driver allows user-level processes to request physical memory, it can still effectively isolate them with the help of the dynamic PMC mechanism. Third, the registers and instructions introduced by DV-Ext cannot be exploited by user-level code to attack the kernel. The *hu\_er* and *hu\_einfo* registers provide information related to DVE and do not leak any data from the host kernel. The *hu\_vitr*, *hu\_vcpuid*, *hu\_vpc*, and *hu\_ehb* reg-

isters, as well as *HUFLUSHGPA*, only control VM behaviors and have no effect on the host kernel. The *HURET* instruction and DVE implement mode switches between the HU mode and a VM, but cannot directly enter the HK mode.

## 8 Performance Evaluation

We answer the following four questions in this section:

**Q1:** How does the DuVisor compare to the KVM/QEMU in terms of hypervisor primitive cost? (§8.2)

**Q2:** How does the performance of applications running on DuVisor compare to that of KVM/QEMU? (§8.3)

**Q3:** What is the performance impact of DV-Ext on the co-located KVM/QEMU that does not use this extension? (§8.4)

**Q4:** How much performance impact does the extended PMP mechanism have on DuVisor’s performance? (§8.5)

### 8.1 Experimental Setup

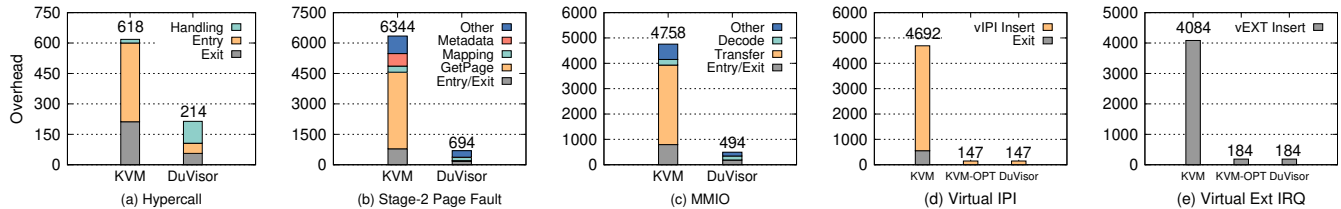
We ran experiments on the cycle-accurate FireSim platform [59], which consists of two FPGA boards. Each FPGA board has eight RISC-V processors (3.2GHz, rv64imafdch), 16GB RAM, and 115GB storage. We created a local area network (LAN) between the two boards using 1Gbps IceNICs for network-related benchmarks. Both FPGA boards were controlled by an EC2 instance running CentOS 7.6.1810 on a 16-core Intel E5-2686 v4 CPU (2.3 GHz) and 240 GB RAM. We used OpenSBI v0.8 [31] as the firmware for RISC-V, and used Linux kernel 5.10.26 as the host kernel, which was equipped with the DV-driver. The baseline is KVM [23] (with H-Ext [35] support) and *QEMU* v7.0.0-rc0 running on Linux 5.16.

For **Q1** and **Q2**, posted interrupts can greatly improve the performance in interrupt-intensive scenarios, but the current open-sourced RISC-V hardware does not support this feature. To make a fair performance comparison, we extended DV-Ext’s interrupt virtualization support to the kernel level and implemented an optimized KVM/QEMU (“**KVM-OPT**”) that enables kernel-level posted interrupts. To answer **Q3**, we compared KVM with a slightly modified KVM (“**KVM-DVext**”) that was patched with context save/restore code related to DV-Ext and virtualization registers. For **Q4**, we compared DuVisor with a PMP-less version (“**DuVisor-noPMP**”) that removes PMP checking from the hardware.

### 8.2 Microbenchmarks

In this section, we quantify the performance of five frequently used hypervisor primitives. We leveraged the *cycle* CSR to measure CPU cycles. Figure 6 shows the average cost of the five operations in KVM and DuVisor. We calculated the average cycle count after recording the total time spent performing each operation 10,000 times. For three synchronous exceptions, we only compared DuVisor with KVM because there is no difference between KVM and KVM-OPT when interrupt virtualization is not involved. For the other two asynchronous exceptions, results of KVM, KVM-OPT





**Figure 6: Breakdown of different hypervisor primitives (Unit: cycles).** (a) shows a *null* hypercall. *Exit*: from invoking a hypercall in the guest VM to arriving at the hypercall handler in the hypervisor. *Entry*: the reverse procedure of *Exit*. *Handling*: processing in the hypercall handler. (b) shows a stage-2 page fault handling. *Entry/Exit*: from triggering a stage-2 page fault in the VM to arriving at the #S2PF handler, and the reverse procedure. *GetPage*: getting the available physical page for the fault GPA. *Mapping*: the PTE update in the stage-2 page table. *Metadata*: maintaining the metadata of the physical page to be mapped. *Other*: other logic such as lock protections and fault GPA checking. (c) shows an MMIO emulation. *Entry/Exit*: from invoking an MMIO operation in the VM to arriving at the user-space MMIO handler, and the reverse procedure. *Transfer*: transfers between the kernel in HK mode and the user-space VMM in HU mode, which DuVisor gets rid of. *Decode*: decoding the corresponding virtual MMIO device according to the fault address. *Other*: other logic, such as checking if the fault address belongs to the MMIO address range. (d) shows a virtual IPI sending. *vIPI Insert*: For KVM, from the hypervisor inserting the virtual IPI and kicking the receiver vCPU to the receiver vCPU's arriving at the IPI handler. For KVM-OPT and DuVisor, from the sender vCPU's writing *hu\_vitr* register to the receiver vCPU's arriving at the IPI handler. *Exit*: Only for KVM, from the sender vCPU's invoking *SEND\_IPI* hypercall to the hypervisor's insertion, and from the receiver vCPU's being kicked to it being inserted with the pending virtual IPI. (e) shows an I/O notification sending. *vEXT Insert*: For KVM-OPT and DuVisor, from the I/O thread's writing *hu\_vitr* register to the vCPU thread's arriving at the IRQ handler. For KVM, from the I/O thread invoking the *SET\_INTERRUPT* interface to the receiver vCPU's arriving at the IPI handler.

and DuVisor are shown.

In the hypercall microbenchmark, both KVM and DuVisor ran a guest VM with a single vCPU pinned to a pCPU. The guest VM invoked a *null* hypercall, which trapped to the hypercall handler and then returned immediately without doing any functional operations. The number of cycles between the start of the hypercall and its return position was counted. As shown in Figure 6-a, DuVisor consumes 65.37% (404 cycles) less time during the hypercall procedure than KVM. This is because DuVisor in the user space does not need to perform operations that are only necessary in the kernel (e.g., enabling and disabling preemption and interrupts).

For stage-2 page fault handling, each hypervisor ran a guest VM with a single vCPU pinned to a pCPU. The guest VM read one byte from a page unmapped in the stage-2 page table, triggering a stage-2 page fault exception trapped to the hypervisor. The hypervisor allocated memory and established a valid mapping in the stage-2 page table before resuming the vCPU execution. We collected cycles before and after the guest VM read. As shown in Figure 6-b, DuVisor spends about 89.06% (5,650 cycles) less time compared with KVM. The main reason for the decreased cycles is that the KVM implementation is generic but more complex, whereas DuVisor can choose a dedicated but more concise implementation. According to our breakdown of the stage-2 page fault handling in KVM, most of the time is spent on getting the available physical page for the guest fault address, accounting for about 59.52% (3,776 cycles) of the total time as the *GetPage* part shows. Similarly, the *Other* and *Metadata* parts in KVM account for 23.31% (1,479 cycles) of the whole process, consisting of many generic Linux kernel logic, such as finding *virtual memory area (VMA)*, taking locks of *mmap* and maintaining metadata in Linux page structures. In com-

parison, DuVisor only spends 26 cycles in the *GetPage* part and 324 cycles in the *Other* and *Metadata* part.

For MMIO emulation, a single-vCPU guest VM pinned to a pCPU performed a load operation from an MMIO address of a virtual console device, which trapped to the user-level hypervisor and immediately returns. We counted the elapsed cycles of the MMIO read operation. The result shows that DuVisor takes 89.62% (4,264 cycles) less time than KVM primarily due to the shorter path of MMIO handling as shown in Figure 6-c. Traditional hypervisors such as KVM offload most MMIO emulations to user mode for security and reliability, which leads to a longer path than DuVisor. The breakdown shows that 65.89% (3,135 cycles) of the time during the MMIO emulation in KVM is spent on multiple world switches: VM(V)  $\leftrightarrow$  KVM(HK)  $\leftrightarrow$  QEMU(HU).

For vIPIs, both KVM and DuVisor ran a dual-vCPU guest VM and pin two vCPUs to separate cores. The sender vCPU sent an IPI and waited for the ACK from the receiver vCPU by polling on the shared memory, while the receiver vCPU wrote to the shared memory as soon as entering the IPI handler to inform the sender. We calculated the cycles on the sender vCPU from sending IPI to getting ACK from the shared memory. Figure 6-d shows the results. Since both KVM-OPT and DuVisor leverage hardware posted interrupt support, their virtual IPI processes are done without hypervisor involvement and thus equally cost 147 cycles. In contrast, the KVM spends 4,692 cycles on sending a vIPI. To send an IPI, the sender vCPU has to invoke a hypercall and trap to the KVM (*Exit* part), which occupies 11.83% of the total cost. While the rest 88.17% is cost by the cumbersome *vIPI Insert* part, in which the hypervisor sends the vIPI request, kicks the receiver vCPU, and inserts the vIPI before resuming the receiver vCPU.



For virtual external interrupts, a single-vCPU guest VM pinned to a pCPU spun and waited for external interrupts. We calculated the average consumed cycles from the hypervisor inserting a virtual external interrupt to the vCPU acknowledging the inserted interrupt in the interrupt handler. Using the same hardware posted interrupt support, both KVM-OPT and DuVisor averagely spend 184 cycles. While KVM needs 4,084 cycles in total due to the long emulation processes, such as kicking the vCPU.

### 8.3 Application Benchmarks

**Table 7: Descriptions of application benchmarks.**

| Name      | Description                                                                                                                                                                                                             |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Netperf   | Netserver v2.6.0 on the local server (guest VM) and Netperf v2.6.0 on the remote client (native) to test the TCP stream throughput for 5 seconds.                                                                       |
| iperf3    | iperf v3.9 on both the local server (guest VM) and the remote client (native) to test the TCP throughput for 10 seconds.                                                                                                |
| Memcached | Memcached v1.6.10 running the memtier benchmark 1.3.0 on the remote client to test transactions per second. The thread number is set to the same as the number of server vCPUs. Each round of the test lasts 5 seconds. |
| Hackbench | Hackbench using Unix domain sockets and default 10 process groups running in 100 loops, measuring the time cost.                                                                                                        |
| CPUPrime  | CPU test in sysbench v0.4.12 that calculates prime numbers up to the max prime 10000. The thread number is set to the same as the number of server CPUs.                                                                |

In this section, we evaluated the performance of five application benchmarks described in Table 7, compared KVM-OPT and DuVisor’s results with native, and analyzed the reasons for the performance differences. We ran the benchmark in three VMs with 1, 2 and 4 vCPUs, respectively. Every VM was equipped with 512MB memory, a virtio-based network device, and a virtio-based block device. In each case, we assigned the same number of CPUs and memory size to native as to VMs via *maxcpus* and *mem* using the kernel command line. To demonstrate the best performance of the in-kernel KVM, we used vhost-net as the network backend of the KVM-OPT VMs. For DuVisor VMs, we implemented a lightweight user-space network backend driver by porting *ixy* [51], a 1,000-LoC user-space ixgbe driver, to FireSim’s IceNet. Each vCPU and I/O thread of a guest VM was pinned to a separate physical CPU to avoid instability caused by the host kernel scheduler. All applications were evaluated after warmup to eliminate stage-2 page faults during benchmarks.

As shown in Figure 7-a and Figure 7-b, both KVM-OPT and DuVisor make full use of the NIC’s bandwidth with different vCPU numbers and have no significant overhead compared to native. Because KVM-OPT and DuVisor used different network backends, their performance when running network-intensive applications can vary due to backend implementations. Nevertheless, what we intend to demonstrate is that DuVisor can attain comparable performance to KVM-OPT’s mature vhost-net with a simply-implemented user-space network backend. For the network-intensive memcached application shown in Figure 7-c, DuVisor has a similar performance to KVM-OPT. However, they introduce up to 35% virtualization overhead when compared with the native. According to our analysis, the reasons for the perfor-

mance degradation mainly consist of the longer network data transfer path and the sub-optimal frontend/backend notifications. Massive small memcached requests travel longer than native before reaching the memcached in VMs due to the I/O virtualization. Besides, guest VMs’ interrupt frequency during the benchmark is much higher than that of the native due to the frequent notifications from the backend drivers, making memcached threads in VMs have less CPU time to process requests. We also compared QEMU/KVM with native on Intel and ARM platforms with similar configurations and find that they also introduce 15% to 40% overhead.

As shown in Figure 7-d, DuVisor is also comparable to KVM-OPT and native for hackbench. It is worth noting that KVM, which did not use the hardware interrupt virtualization, will incur about 12% more overhead in this experiment. The reason is that many IPIs are generated under this test, and the virtual IPI operation can be effectively accelerated by the posted interrupt, as shown in Figure 6-d. This also explains why DuVisor’s better microbenchmark performance results in no better application performance than KVM-OPT. Infrequent VM exits in DuVisor and KVM-OPT result in very low costs for hypervisor primitives (<5% CPU cycles), which are hardly observable in application benchmarks. Figure 7-e indicates that KVM-OPT and DuVisor attain the same performance as native execution in the CPUPrime benchmark.

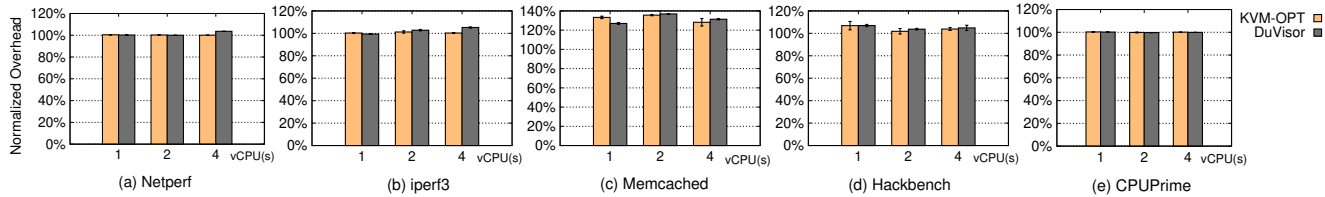
As a result, the design of DuVisor does not introduce performance overhead compared with KVM-OPT, while achieving better host kernel security and reliability.

### 8.4 Impact on Co-located KVM VMs

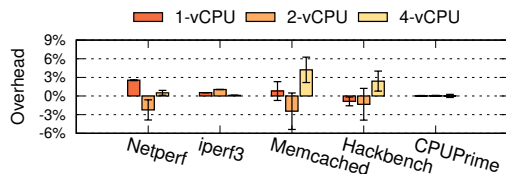
When co-locating a traditional in-kernel hypervisor together with a user-level hypervisor, both hypervisors can independently configure registers related to virtualization, which can lead to VM state conflicts if not handled properly. Therefore, the host kernel needs to save and restore virtualization-related registers during context switches between them, introducing additional switching latency. To clarify how much impact such delay has on KVM, we evaluated and compared the application performance of KVM and KVM-DVext (with necessary context save/restore code). Figure 8 shows that there is no discernible performance difference between KVM and KVM-DVext, indicating that such additional switching latency due to DuVisor’s co-location has little impact on traditional VMs.

### 8.5 Memory Virtualization Overhead

**Scaling Memory:** To show DuVisor’s memory scalability compared with KVM-OPT, we ran memcached in a 4-vCPU guest VM with 512MB, 1024MB, 1536MB and 2048MB memory. As shown in Figure 9-a, DuVisor achieves almost the same performance as KVM-OPT in all cases. Compared with KVM-OPT, the memory virtualization of DuVisor differs only in that it places the stage-2 page table configuration in the user space and extends PMP for security checks. There-



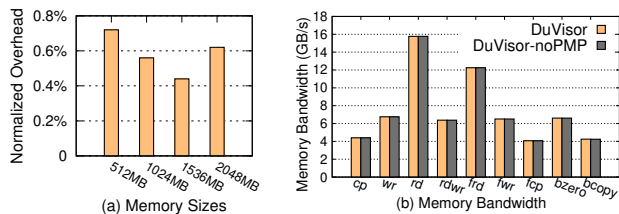
**Figure 7: Normalized VM performance of real-world applications running in KVM-OPT and DuVisor.** The Y-axis is the normalized overhead compared with the corresponding native environment. Error bars are added due to the performance fluctuation.



**Figure 8: Normalized performance overhead of real-world applications of KVM-DVext compared with KVM.**

fore, there is little impact on the memory access latency and thus scales well as the memory size grows.

**Impact of PMP:** DuVisor slightly extends the PMP hardware with a V bit to verify the validity of memory accesses from guest VMs. To evaluate whether this memory protection mechanism degrades the memory performance of the VM, we compared the memory test results of *lmbench* between DuVisor and DuVisor-noPMP in a dual-vCPU VM with 512MB memory. As shown in Figure 9-b, the memory bandwidth of the VM is almost the same with and without PMP checking. The result shows that the design of PMC does not introduce significant overhead to VMs.



**Figure 9: Performance of DuVisor's memory virtualization.** (a) DuVisor's normalized overhead of different sizes of memory compared with KVM-OPT using memcached. (b) DuVisor's memory performance w/ and w/o PMC using *bw\_mem* of *lmbench*.

## 9 Discussion and Limitations

**Nested Virtualization.** In traditional nested virtualization [41, 71, 77, 92], VM exits are intercepted by the L0 hypervisor (bare-metal one) before being handled by the L1 hypervisor (nested one), incurring tremendous runtime overhead. Prior work [72] optimizes nested VM exits via paravirtualization and passthrough. In the future, we plan to extend the idea of DVE to optimize nested virtualization by directly trapping VM exits to the L1 hypervisor.

**Memory Utilization.** DuVisor's PMP-based memory virtualization may lead to low memory utilization. To cover all VM memory with a limited number of PMP regions, we use the coarse-grained physical memory protection that can result in memory fragmentation and scalability issues. Addi-

tionally, it is difficult to support memory multiplexing (e.g., deduplication and overcommitment) among VMs based on PMP. Fortunately, such limitations can be mitigated by memory migration software mechanisms [65]. They can also be resolved through scalable fine-grained memory protection hardware mechanisms. For example, the RISC-V PMP table [25] has been proposed recently, which extends PMP to support physical memory restriction in page granularity.

**IOMMU Support.** Although IOMMU [29] is not yet supported on currently available RISC-V hardware, DuVisor is theoretically able to support it. Specifically, the stage-2 page table of IOMMU can be directly controlled by DuVisor, as the DV-driver can restrict access from guest devices by the IOPMP mechanism [30] with the V bit introduced by DV-Ext. The passthrough of the guest devices and the management of IOMMU's stage-1 page table within VMs is no different from traditional virtualization.

**DV-Ext Universality.** While the current prototype is implemented on RISC-V platforms, applying DV-Ext to other architectures would also be feasible, as they all share the same high-level virtualization functions. Consider Intel VMX hardware virtualization as an example.

*CPU virtualization:* Intel VMX directs all VM exits to the host kernel mode, and guest states in the in-memory VMCS can only be obtained and configured with the kernel-only VMREAD and VMWRITE instructions. To apply DV-Ext, Intel can just take VM exits and expose such VMX instructions to the DuVisor in the host user mode, while preventing access to host states (e.g., host CR3) in the VMCS.

*Memory virtualization:* The stage-2 page tables can be managed by the user-mode DuVisor without hardware modifications. However, PMP-like primitives are also necessary to enforce security. Fortunately, such hardware has emerged with confidential VM extensions [1, 21, 64, 69]. To apply DV-Ext, Intel can extend existing TDX's Physical Address Metadata Table (PAMT) [47] to provide fine-grained physical memory protection for DuVisor.

*Interrupt virtualization:* Intel can expose VMCS fields related to virtual interrupts to the DuVisor to deliver virtual interrupts. Specifically, virtual interrupts can be issued in user space by writing the VMCS with the user-mode VMX instructions mentioned above.

**Host Kernel Vulnerability.** Although DuVisor minimizes the attack surfaces of in-kernel hypervisors that are exposed to VMs, it does not completely exclude the host kernel from the VMs' runtime. Resource management (e.g., schedul-

ing and physical interrupt handling) involves non-hypervisor components in the host kernel, which may still contain many vulnerabilities, especially in mainstream monolithic kernels. Consequently, DuVisor is still vulnerable to the vulnerabilities of non-hypervisor components. Reconstructing it as microkernels may mitigate the problem, however, there are tradeoffs among compatibility, performance, and security.

## 10 Related Work

**Moving Kernel Functions to Userspace.** Deprivileging kernel features to userspace is a classic approach to enhance security, ease development, and improve performance. Microkernel is one typical design [52,62,70,78], where system services such as file systems and drivers run in user mode. For monolithic kernels, similar methods also exist, which implement the file systems [50,79], scheduler [57], network service [75] and sandbox [53] in user space. In terms of hypervisor functions, research focuses on reducing the hypervisor TCB by moving some of its functions to userspace, as demonstrated by designs such as DeHype [91] and NOVA [87]. Unlike DuVisor, they still require an in-kernel trusted module to perform VM-plane functions via hardware virtualization interfaces. DuVisor is the first system that entirely moves runtime VM-plane functions to user space, benefiting virtualization architectures on both monolithic kernels [40,61] and microkernels [55].

**Securing VMs.** Apart from the above solutions, many other studies have investigated how to achieve better isolation for VMs atop unreliable hypervisors. Numerous efforts have focused on reducing the hypervisor TCB [66,81,85,92]. CloudVisor [92] leverages nested virtualization to deprivilege the Xen [40] hypervisor. HypSec [66] separates a tiny CoreVisor as TCB from the KVM hypervisor. Other approaches have worked on hardening the hypervisor TCB, such as SeKVM [67,68], which use formal verification to ensure security guarantees of hypervisors. Unlike DuVisor, these solutions still rely on in-kernel hardware virtualization interfaces and incur modest performance overhead compared to unmodified traditional hypervisors.

Some solutions improve hypervisor reliability by providing per-VM hypervisor instances that are isolated from each other. Nexen [84] deconstructs the hypervisor into per-VM non-privileged service slices. HyperLock [90] decomposes the hypervisor into isolated shadow copies for each VM. In contrast to DuVisor, they still rely on traditional hardware virtualization interfaces and suffer from performance penalties of software isolation mechanisms. Others propose hardware extensions to remove the vulnerable hypervisor from the TCB [39,58,60,66,89]. NoHype [60] eliminates the hypervisor and its attack surfaces by static partitioning physical resources with hardware modifications. Nonetheless, it disallows resource oversubscription and is thus impractical for deployment in production scenarios.

Industrial confidential virtual machine (CVM) solutions,

such as AMD SEV-SNP [1] and ARM CCA [5,69], leverage specialized hardware security extensions to protect the data of VMs against a malicious hypervisor, which cannot access or taint the memory and registers of VMs. Both CVMs and DuVisor are vulnerable to non-hypervisor DoS attacks because they both rely on the host kernel. However, unlike CVMs, DuVisor can avoid DoS attacks due to in-kernel hypervisor vulnerabilities by deprivileging all VM-plane functions to user space to minimize the host kernel's runtime attack surfaces exposed to VMs. On the other hand, CVMs require guest OS device driver modifications, while DuVisor supports unmodified VMs.

CVM and DuVisor are orthogonal techniques that can be combined for greater benefits. The design of DuVisor can be applied to defend against DoS attacks due to in-kernel hypervisor vulnerabilities for CVMs, which we plan to explore as future work. Existing CVMs are controlled by the in-kernel hypervisor through secure firmware interfaces (e.g., ARM CCA's RMM and TF-A, Intel TDX module) that can only be invoked in the host kernel. DuVisor can be used alongside CVMs by exposing these interfaces to user space, thereby eliminating shared in-kernel hypervisor vulnerabilities.

## 11 Conclusion

We introduce the first delegated virtualization architecture that delegates all VM-plane virtualization functions to user space without trapping into the host kernel, minimizing attack surfaces exposed to VMs by in-kernel hypervisor components. To enable delegated virtualization, we present DV-Ext and DuVisor. DV-Ext is a hardware virtualization extension that securely exposes hardware virtualization interfaces to user space. DuVisor is a user-level hypervisor design that directly serves VM-hypervisor interactions in user space. We also present security techniques to prevent malicious use of DV-Ext. We have implemented a prototype for DV-Ext and DuVisor on the RISC-V platform. The security and performance evaluation results demonstrate that DuVisor protects the host kernel from hypervisor vulnerabilities without compromising performance compared to Linux KVM, establishing a new direction for secure virtualization research and development.

## 12 Acknowledgments

We express our sincere gratitude to our shepherd Jason Nieh for providing us with valuable suggestions that significantly helped in improving our paper. We thank the anonymous OSDI reviewers for their insightful suggestions. We are grateful to Yubin Xia and Rong Chen for their thorough and constructive comments that greatly improved this paper. We also thank Chenhui Ji and Yifan Tan for their contributions to the artifact evaluation. This work was supported in part by the National Natural Science Foundation of China (No. 62002218, 61925206, 62132014).

## References

- [1] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>. Referenced May 2023.
- [2] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. <https://www.amd.com/system/files/TechDocs/24593.pdf>. Referenced May 2023.
- [3] An EPYC Escape Case Study of KVM. <https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html>. Referenced May 2023.
- [4] ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. <https://developer.arm.com/documentation/102105/latest>. Referenced May 2023.
- [5] ARM CCA Hardware Architecture. <https://developer.arm.com/documentation/ddi0615/latest/>. Referenced May 2023.
- [6] CVE-2013-1796. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1796>. Referenced May 2023.
- [7] CVE-2014-0049. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0049>. Referenced May 2023.
- [8] CVE-2014-7842. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7842>. Referenced May 2023.
- [9] CVE-2018-16882. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16882>. Referenced May 2023.
- [10] CVE-2019-19332. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19332>. Referenced May 2023.
- [11] CVE-2019-6974. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974>. Referenced May 2023.
- [12] CVE-2019-7221. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1760>. Referenced May 2023.
- [13] CVE-2021-22543. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22543>. Referenced May 2023.
- [14] CVE-2021-29657. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29657>. Referenced May 2023.
- [15] CVE-2021-4093. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4093>. Referenced May 2023.
- [16] CVE-2021-43056. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43056>. Referenced May 2023.
- [17] CVE-2021-8106. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8106>. Referenced May 2023.
- [18] DPKD. <https://www.dpkd.org/>. Referenced May 2023.
- [19] Intel® 64 and IA-32 Architectures Software Developer's Manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf>. Referenced May 2023.
- [20] Intel® Architecture Instruction Set Extensions Programming Reference. <https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html>. Referenced May 2023.
- [21] Intel® Trust Domain Extensions (Intel® TDX). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>. Referenced May 2023.
- [22] KVM CVE. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=KVM>. Referenced May 2023.
- [23] KVM RISC-V. <https://github.com/KVM-riscv>. Referenced May 2023.
- [24] Linux Seccomp. <https://en.wikipedia.org/wiki/Seccomp>. Referenced May 2023.
- [25] PMP Table Extension. <https://docs.google.com/document/d/158j99tm1gmZ5VH010scZhLKRU51JzJhj2zS2R1UWMeQ/edit#heading=h.rjobwmo1vft1>. Referenced May 2023.
- [26] QEMU: A Generic and Open Source Machine Emulator and Virtualizer. <https://www.qemu.org/>. Referenced May 2023.
- [27] QEMU-KVM Guest to Host Kernel Escape Vulnerability: vhost/vhost\_net kernel buffer overflow. [https://bugs.gentoo.org/show\\_bug.cgi?id=CVE-2019-14835](https://bugs.gentoo.org/show_bug.cgi?id=CVE-2019-14835). Referenced May 2023.
- [28] RISC-V Hypervisor Extension, Version 1.0.0-rc. <https://github.com/riscv/riscv-isa-manual/blob/master/src/hypervisor.tex>. Referenced May 2023.
- [29] RISC-V IOMMU Specification. <https://github.com/riscv-non-isa/riscv-iommu/blob/main/riscv-iommu.pdf>. Referenced May 2023.
- [30] RISC-V IOPMP Specification. <https://github.com/riscv-non-isa/iopmp-spec>. Referenced May 2023.
- [31] RISC-V OpenSBI, Version 0.8. <https://github.com/riscv-software-src/opensbi/releases/tag/v0.8>. Referenced May 2023.
- [32] RISC-V Privileged Architectures, Version 1.12. <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>. Referenced May 2023.
- [33] RISC-V "N" Standard Extension for User-Level Interrupts, Version 1.1. <https://five-embeddev.com/riscv-isa-manual/latest/n.html>. Referenced May 2023.



- [34] Rocket Chip. <https://github.com/chipsalliance/rocket-chip>. Referenced May 2023.
- [35] Rocket Chip H-Ext PR. <https://github.com/chipsalliance/rocket-chip/pull/2841>. Referenced May 2023.
- [36] The Architecture of VMware ESXi. [https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/ESXi\\_architecture.pdf](https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/ESXi_architecture.pdf). Referenced May 2023.
- [37] The Current State of Kernel Page-table Isolation. <https://lwn.net/Articles/741878/>. Referenced May 2023.
- [38] XEN CVE. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=XEN>. Referenced May 2023.
- [39] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy in-Context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, page 38–49, New York, NY, USA, 2010. Association for Computing Machinery.
- [40] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP '03*, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.
- [41] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, page 423–436, USA, 2010. USENIX Association.
- [42] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 31(5):143–156, oct 1997.
- [43] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4), nov 2012.
- [44] J. P. Buzen and U. O. Gagliardi. The Evolution of Virtual Machine Architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition, AFIPS '73*, page 291–299, New York, NY, USA, 1973. Association for Computing Machinery.
- [45] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and Xiaofeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18*, page 601–608, New York, NY, USA, 2018. Association for Computing Machinery.
- [46] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP '11*, page 189–202, New York, NY, USA, 2011. Association for Computing Machinery.
- [47] Intel Corporation. Architecture Specification: Intel Trust Domain Extensions (Intel TDX) Module, Section 13. 2020.
- [48] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.
- [49] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.
- [50] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.
- [51] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User Space Network Drivers. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*, September 2019.
- [52] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII*, page 168–178, New York, NY, USA, 2008. Association for Computing Machinery.
- [53] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004.
- [54] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.
- [55] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Workshop on Systems, APSys '10*, page 19–24, New York, NY, USA, 2010. Association for Computing Machinery.
- [56] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A Case against (Most) Context Switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21*, page 17–25, New York, NY, USA, 2021. Association for Computing Machinery.
- [57] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOST: Fast &

- Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.
- [58] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 272–283, New York, NY, USA, 2011. Association for Computing Machinery.
- [59] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 29–42. IEEE Press, 2018.
- [60] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 350–361, New York, NY, USA, 2010. Association for Computing Machinery.
- [61] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.
- [62] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
- [63] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [64] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *Proceedings of the 2023 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '23, Boston, MA, July 2023. USENIX Association.
- [65] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 638–654, New York, NY, USA, 2021. Association for Computing Machinery.
- [66] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1357–1374, Santa Clara, CA, August 2019. USENIX Association.
- [67] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, August 2021.
- [68] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799, 2021.
- [69] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the ARM Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.
- [70] J. Liedtke. On Micro-Kernel Construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, dec 1995.
- [71] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 201–217, New York, NY, USA, 2017. Association for Computing Machinery.
- [72] Jin Tack Lim and Jason Nieh. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 557–574, New York, NY, USA, 2020. Association for Computing Machinery.
- [73] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.
- [74] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.
- [75] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.
- [76] Zeyu Mi, Haibo Chen, Yinqian Zhang, Shuanghe Peng, Xiaofeng Wang, and Michael K. Reiter. CPU Elasticity to Mitigate Cross-VM Runtime Monitoring. *IEEE Transactions on Dependable and Secure Computing*, 17(5):1094–1108, 2020.

- [77] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In Srdjan Capkun and Franziska Roesner, editors, *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1695–1712. USENIX Association, 2020.
- [78] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019*. Association for Computing Machinery.
- [79] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas E. Anderson. High Velocity Kernel File Systems with Bento. In Marcos K. Aguilera and Gala Yadgar, editors, *Proceedings of the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 65–79. USENIX Association, 2021.
- [80] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '08*, page 151–160, New York, NY, USA, 2008. Association for Computing Machinery.
- [81] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. Delusional Boot: Securing Hypervisors without Massive Re-Engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 141–154, New York, NY, USA, 2012. Association for Computing Machinery.
- [82] Gaoning Pan, Xingwei Lin, Xinlei Ying, Jiashui Wang, and Chunming Wu. Scavenger: Misuse Error Handling Leading To QEMU/KVM Escape. In *Black Hat Asia, 2021*.
- [83] L. H. Seawright and R. A. MacKinnon. VM/370—A study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.
- [84] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
- [85] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: A Thin Hypervisor for Enforcing i/o Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09*, page 121–130, New York, NY, USA, 2009. Association for Computing Machinery.
- [86] Baibhav Singh and Rahul Kashyap. Back To The Future: A Radical Insecure Design of KVM on ARM. In *Black Hat USA, 2018*.
- [87] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys '10*, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery.
- [88] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.
- [89] Jakub Szefer and Ruby B. Lee. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, page 437–450, New York, NY, USA, 2012. Association for Computing Machinery.
- [90] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 127–140, New York, NY, USA, 2012. Association for Computing Machinery.
- [91] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.
- [92] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP '11*, page 203–216, New York, NY, USA, 2011. Association for Computing Machinery.

## A Artifact Appendix

### Abstract

The artifact evaluation of DuVisor contains two parts: the security evaluation and the performance evaluation. For security evaluation, we evaluate representative CVEs in the DuVisor on the QEMU-emulated RISC-V platform. For performance evaluation, we measure the performance of various microbenchmarks and application benchmarks on native, DuVisor, vanilla KVM and optimized KVM using the cycle-accurate FireSim platform.

### Scope

**Security Evaluation:** DuVisor is able to prevent host kernel from crashing even if the user-level VM-plane is attacked. As mentioned in the Table 6 of our paper, this artifact emulates 6 representative KVM CVEs and evaluates their impact on the system. The results can show that these CVEs could crash DuVisor itself, but the host kernel can continue to execute other programs (including DuVisor VMs) normally.

**Performance Evaluation:** DuVisor achieves higher security while also maintains comparable performance to the optimized KVM (i.e., KVM-OPT in our paper). As shown in Figure 7-10 of our paper, this artifact compares various benchmarks between DuVisor and KVM, and also evaluates the impact of DV-Ext hardware extension KVM.

The results can show that DuVisor performs good and DV-Ext has little impact on existing KVM.

## Contents

- **Run-time environment:** FireSim cycle-accurate FPGA platform based on AWS EC2 instances (two C5 and one F1)
- **Hardware:** QEMU (security AE) and RocketChip (performance AE)
- **Software:** OpenSBI, Linux, QEMU, DuVisor, related benchmarks
- **Metrics:** Benchmark results, usually latency and throughput
- **Estimated time:** about 20 hours with pre-built software images

- **Available on GitHub:** <https://github.com/IPADS-DuVisor/ae-guide/tree/main/>

## Hosting

The artifacts are available on the GitHub, please refer to the main branch of this guide: <https://github.com/IPADS-DuVisor/ae-guide/tree/main/>

## Requirements

Because the FireSim platform relies on special AWS FPGA (F1) instances, requiring multiple machines and complicated environment configurations. Besides, a newer version of FireSim platform may not be compatible with an older one. To simplify the AE procedure, we provided pre-configured AWS instances for reviewers.







# Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud

Ziqiao Zhou\* Yizhou Shan<sup>†</sup> Weidong Cui\* Xinyang Ge\*<sup>‡</sup> Marcus Peinado\* Andrew Baumann\*<sup>§</sup>

\*Microsoft Research

<sup>†</sup>University of California, San Diego

## Abstract

Virtual machines are the basis of resource isolation in today’s public clouds, yet the security risks of entrusting that isolation to a cloud provider’s hypervisor are substantial. Such concerns have motivated hardware extensions for “confidential VMs” that seek to remove the hypervisor from the trusted computing base by adding a highly-privileged firmware layer that checks hypervisor actions, and supports memory encryption and remote attestation. However, the hypervisor retains control of resource management and observes associated guest actions including nested page table faults and CPU scheduling, and thus confidential VMs remain vulnerable to an ever-changing variety of hypervisor-level side channel attacks. Bare-metal cloud servers avoid such leaks, but remain a niche due to the high cost of dedicated hardware.

We observe that typical cloud VMs run with a static allocation of memory and discrete cores, and increasingly rely on I/O offload, thus negating the apparent need for a hypervisor and the fragile hypervisor/guest isolation boundary. Our design, *core slicing*, enables multiple untrusted guest OSes to run on shared bare-metal hardware. To ensure isolation without the complexity of virtualization, we propose simple hardware extensions that restrict guests to a static *slice* of a machine’s cores, memory and virtual I/O devices, and delegate resource allocation to a dedicated management slice. We demonstrate practicality and evaluate performance with prototypes for RISC-V and x86.

## 1 Introduction

We are in the early stages of a new generation of trusted execution environment (TEE). Motivated by cloud workloads, their main new feature is the ability to run “*confidential VMs*” inside the TEE [13, 17, 53]. Driven by the demand for secure cloud computing in which the user need not trust the cloud

provider [78], these TEEs are expected to see much wider adoption than their predecessors [50]. Although confidential VMs offer enhanced functionality, their security model and architecture are largely identical to earlier TEEs such as SGX. The user shares one or more processor cores with a powerful adversary who controls hardware resources. Processor extensions provide the TEE with private memory and a trusted “context switch” to prevent the administrator-adversary from directly breaking the confidentiality and integrity of the TEE.

However, the past decade has produced a broad and rapidly growing spectrum of attacks on this model [22, 25, 41, 44, 62, 65, 67, 68, 72, 82, 90, 98, 104, 110, 112–116], including the complete breakdown of SGX security on several occasions [112, 114, 115]. Most of these can be described as side-channel attacks. As §2.3 will describe, they take advantage of the fact that attacker and victim run on the same core and share a multitude of sometimes obscure microarchitectural components. The same risks [39, 44, 67, 68] exist in confidential VMs. Although today’s confidential VM architectures remove privileges from the host hypervisor (e.g., the ability to read plaintext guest memory), it retains a large degree of control over guest execution, such as the ability to arbitrarily interrupt guests, leading inexorably to side-channel attacks.

It is this paper’s thesis – backed by evidence from recent work on computer architecture [31, 96, 97] and security [60] – that these failures of TEE hardware are not isolated events of the past. More than a decade of microarchitectural optimizations have taken processor complexity to a level where it is practically impossible to reason about isolation boundaries within a core [60, 96]. This problem is not only the likely ultimate root cause of the known attacks, but it is also bound to result in periodic breakdowns of TEE security for the foreseeable future. Indeed, Intel takes the position that side channels “can’t be eliminated” [48, 51, 56] but that it will provide mitigations as new vulnerabilities are found. While similar to the current approach to software vulnerabilities, this expects the users of confidential cloud computing (e.g., financial institutions and governments) to tolerate data leaks whenever a new side channel is discovered.

<sup>†</sup>Yizhou Shan is now at Huawei Cloud.

<sup>‡</sup>Xinyang Ge is now at Databricks.

<sup>§</sup>Andrew Baumann is now at Google.

In this paper, we present a much more robust TEE architecture, *core slicing*, that is realistic for infrastructure-as-a-service (IaaS) workloads. Rather than making confidential guests share cores with an adversarial hypervisor, we give the guest exclusive access to its own CPU cores. This moves the isolation boundary to the much more defensible and robust one between processor cores. We show that this boundary can be enforced with simple (and thus less fragile) hardware.

We observe that, although cloud guests may benefit from a VM-level execution abstraction, cloud providers do not exploit the full complexity enabled by hypervisor-based virtual machines for IaaS workloads. For example, although hypervisors support time-slicing VMs on shared cores, VMs offered by major public cloud providers including Amazon [11] and Azure [81] are sized at core granularity and scheduled on distinct physical cores [7, 76, 77]. Likewise, the memory allocated to guest VMs is static; techniques such as memory ballooning [118] or transparent page sharing [118, 124] are avoided. Cloud providers are also moving to reduce the overhead of I/O virtualization by offloading I/O processing to dedicated hardware [5, 8, 36]. To ensure that resources sold match those available, cloud providers limit oversubscription to only their own (first-party) VMs [28] or disable it entirely [7]. Overall, although today's cloud runs virtual machines, leading public cloud providers do so using an effectively static allocation of cores and memory. The hypervisor is relied upon for isolation, but it does so merely by partitioning platform resources.

By giving the guest exclusive access to CPU cores, core slicing eliminates the potential for the entire class of side channels where the attacker shares per-core microarchitectural resources with its victim. Moreover, we enforce this isolation boundary with a new hardware mechanism that is self-contained and simple enough to permit reasoning about confidentiality and integrity. Fully isolated guest OSes (or, potentially, guest hypervisors) run in their own *slice* of a machine. Each slice consists of a dedicated, static allocation of cores, memory, and directly-assigned I/O devices (e.g., the virtual functions of network and storage controllers); hardware ensures that the cores of a slice are sequestered such that they have no access to memory or I/O devices outside the slice, nor can they interrupt cores of other slices. Because *only* a single guest runs code on any given core, a huge class of microarchitectural side-channel leaks are out of scope, and continued innovation in complex microarchitectural performance optimizations is unhindered, since those cannot impact the TEE isolation boundary. While core slicing eliminates intra-core leakage, it does not prevent *cross-core* side channels such as CrossTalk [91] which will have to be addressed by other means. Nevertheless, we believe that obviating intra-core channels removes by far the largest and most serious part of today's side channel problem in terms of the number and seriousness of known attacks, granularity of sharing, and number of shared components.

Resource allocations are determined by a *slice manager* that runs on a dedicated core (ideally, a separate low-power processor), and is responsible for starting and stopping slices, but is otherwise untrusted by the guest. Guest kernels (or guest hypervisors) are enlightened if necessary to run within a slice, ensuring that they attempt only access to those named resources available to them. For example, a guest cannot assume that physical memory starts at address 0, nor that processors have contiguous IDs; in practice, modern kernels including Linux make no such assumptions, and it suffices to pass boot-time information on the accessible resources.

In the following §2, we elaborate on the security risks of confidential VMs, and explore the way VMs are deployed in the cloud today. Like Keller et al. [58] over 10 years ago, we find that hypervisors add significant needless complexity to the cloud's trusted computing base. However, their system No-Hype [107] did not protect VMs from the (fully trusted) cloud provider and relied on virtualization hardware to enforce isolation. The numerous attacks demonstrated since [31, 62, 97] showed that the security provided by this hardware is fragile [62, 96]. By contrast, core slicing maintains a strict separation between core processor logic that is performance-critical and thus complex, and the hardware that enforces isolation, which is not performance critical and simple. It also permits guests to run their own bare-metal hypervisors.

To grant guests bare-metal access to cores in a shared machine while still securely isolating them from one another raises a key design challenge: *without a more privileged software layer on the core, what can enforce isolation?* The key insight behind our design is that simple hardware techniques used to enable trusted execution features such as secure boot and remote attestation for an entire system [127] can be adapted and applied at the granularity of individual cores to help resolve this dilemma. Specifically, §3 contributes *lockable filter registers* and a *core-local secure reset* mechanism, and describes how they can be used to enable core slicing.

To test the practicality of our design, we build two prototypes. The first (§4) leverages RISC-V physical-memory protection (PMP) registers [93, §3.6] to run multiple isolated Linux slices. The other x86-based prototype (§5) lacks security but enables an evaluation (§6) showing that core slicing offers bare-metal performance without VM overheads, with a substantially smaller TCB, while closing side channels based on caches, page faults [128], and other intra-CPU resources. We also analyze traces from a public cloud to find that we can allocate physically-contiguous slice memory, and confirm that our extensions add minimal hardware cost to an existing design. Finally, §7 outlines a path to applying our design to more mainstream architectures, §8 covers related work, and §9 concludes.

## 2 Background and motivation

### 2.1 Hardware-accelerated virtualization

Since VMware first demonstrated the value of virtual machines on commodity platforms [23], hardware vendors have added features to progressively reduce the overhead of virtualization. Although first-generation VM hardware suffered poor performance [1], the gap closed to the point where today's cloud platforms rely fully on hardware support for CPU virtualization, with features such as nested paging and virtual APICs ensuring that many guest VMs now run with low (<5%) CPU overhead compared to bare-metal execution [2]. Unfortunately, this is not true of all VMs; a recent study by Teabe et al. [108] found that up to 30% of CPU time was consumed by virtualization overheads on memory-intensive workloads, even when using nested address translation with huge pages. Other studies reported similar or worse address translation overheads (even with huge pages) [3, 37, 88].

Besides CPU features, modern hardware also supports efficient virtualization of I/O, through advanced IOMMUs [12, 49] and single-root I/O virtualization (SR-IOV) devices [32, 63]. These enable low-overhead virtual I/O by permitting a single physical I/O adapter (such as a network interface or storage controller) to export multiple virtual PCI functions. These are configured by a hypervisor and assigned to guest VMs by installing appropriate IOMMU translations and interrupt mappings. A guest OS thus interacts directly with the device, without the software overhead of traditional I/O virtualization [117]. The hypervisor's role is reduced to configuring the virtual devices and mapping them to the guests, a slower (control path) operation generally performed at startup.

### 2.2 Confidential VMs

Notwithstanding attempts to reduce the size or attack surface of cloud hypervisors [27, 66, 105, 107, 121, 125], the cloud's trusted computing base is controlled by cloud providers and opaque to its users. Threats such as supply-chain attacks [85] and rogue employees (e.g., cloud administrators and developers) have alternatives to traditional cloud architecture [55, 106, 126]: new architecture extensions such as AMD SEV-SNP [13, 78], Intel TDX [53] and Arm Realms [17] seek to remove the hypervisor entirely from the guest's TCB by extending the approach of earlier user-level TEEs such as Intel SGX [50]. In these designs, guest memory and register context are encrypted by hardware, and resource management actions of the hypervisor, such as mapping of memory to the guest, are checked for consistency with the expected VM state. This prevents, for example, a compromised hypervisor from interfering with a guest's memory layout. Finally, like other trusted computing technologies, these designs include a hardware root of trust with support for remote attestation of the guest VMs, enabling a cloud user to verify that their VM has

been correctly launched before trusting it with any secrets.

While the various architectures share many similarities, they differ in the trusted components that check hypervisor operations and enable remote attestation. In AMD's design, these are delegated to firmware on a separate platform security processor, whereas in the Intel and Arm designs these are performed by trusted and attested software running on the CPU itself. Regardless of where it runs, the relevant firmware/software must be trusted by both host and guest, and although it is simpler than a full hypervisor, that is hardly a guarantee of correctness. Notably, AMD's firmware (the only one of the three to have reached production) has already suffered serious vulnerabilities [13, 14, 24, 26].

### 2.3 Side channels in processor-based TEEs

Because confidential VMs inherit from SGX the key design feature of a privileged attacker who controls resources and shares processor time with the TEE, we expect them to remain vulnerable to many forms of side-channel attacks similar to those that devastated SGX [112, 114, 115].

Several attacks have demonstrated that the processor's address translation mechanism can be used to extract information such as cryptographic keys, text documents or JPEG images from SGX enclaves [101, 111, 128]. In these attacks, the adversary manipulates or simply monitors page tables to observe addresses accessed by the victim. While these attacks were demonstrated for SGX, it is clear that they carry over to TEEs like AMD SEV where the attacker controls nested page tables and handles nested page faults.

Other transmission channels include processor caches [22, 41, 82], branch prediction hardware [65] and interrupt latency [90, 113]. Some of these attacks generalize not only beyond SGX but also beyond TEEs. These channels also form the basis for tools that allow the adversary to single-step instruction-by-instruction through the enclave code [110] and to replay TEE instructions arbitrarily many times without having to rerun the TEE code [104]. Both techniques generalize beyond SGX, as they only require the adversary to control address translation and interrupts, respectively.

Speculative execution attacks have been used to leak information across all x86 isolation boundaries, including virtual machines and SGX [25, 62, 72, 98, 112, 114–116]. For example, Foreshadow [112] results in the disclosure of the entire enclave memory and the processor's SGX attestation key. To mitigate such attacks, confidential VMs rely on the same basic approach as SGX: microarchitectural tweaks and microcode patches to the "context switch" path between TEE and host code. More recent research demonstrates that side channels remain a problem on AMD SEV, including SQUIP attacks [39] via scheduler queues within the *same CPU core*; CipherLeaks [68] via *online encrypted memory analysis*; and attacks via *hypervisor-observable nested page faults* [44, 67]. By contrast, core slicing avoids the shared core resources,



the online memory access from other security domains (i.e., slices), and the hypervisor.

## 2.4 VMs as used in public clouds

We next look at how VMs are deployed in clouds today, focusing on infrastructure-as-a-service platforms, which offer VMs backed by guaranteed resources (CPUs, memory, and in some cases accelerators and I/O bandwidth). We consider Amazon EC2 and Microsoft Azure, as they collectively represent 60% of the worldwide IaaS market [38]. We do not consider non-IaaS workloads such as serverless or micro instances for which core slicing may be a poor fit.

**VMs are allocated at core granularity.** Despite offering a plethora of different VM sizes [11], all current-generation VMs in Amazon EC2 occupy at least an entire core (i.e., two vCPUs on platforms that support hyperthreading), and Amazon states explicitly that host cores are “pinned” to specific guest vCPUs and are not shared across guests [7]. Microsoft Azure also offers a wide range of VM configurations [81]. Like Amazon, Azure does not oversubscribe customer vCPUs: Cortez et al. [28] note explicitly that their system will “only oversubscribe servers running first-party workloads.”

For both providers, *burstable VMs* [10, 80, 119] represent the main special case as far as CPU allocation is concerned. These VM types are optimized for workloads that are mostly idle, with only occasional bursts of CPU activity. Like all cloud VMs, they have a fixed number of vCPUs, but consume on average only a fraction of their vCPU allocation in physical CPU runtime. Thus, of all the VM types offered across EC2 and Azure, burstable VMs are the only type that fundamentally requires the use of a hypervisor to perform time-slicing, in order to account for the VM’s actual CPU utilization, and (presumably) to benefit from sharing CPUs across burstable VMs. All other VMs have a guaranteed allocation of physical CPUs, and for these the cloud provider derives no obvious benefit from hypervisor time-slicing.

**Virtual I/O is becoming fully offloaded.** Cloud vendors have deployed dedicated hardware “cards” that replace software I/O virtualization stacks, exposing virtual devices to guests directly via SR-IOV. For example, Amazon Nitro [5, 7] and Azure AccelNet [36] enable low-overhead networking. EC2 also supports direct access to NVMe storage [8], and Azure supports SR-IOV for InfiniBand and GPUs [57]. It thus seems reasonable to assume that, in the near future, the only I/O devices that are still emulated by host software will be those that are not performance sensitive, such as the virtual serial port or console device used for debugging.

**Advanced VM features are not needed.** Cloud providers rely on VMs to isolate tenants, but make little to no use of

the advanced features enabled by full virtualization. Some features are incompatible with the IaaS model of dedicated resources. For example, a customer paying for a VM with 16 GiB of RAM has no incentive to enable memory ballooning [118] and return unused memory to the hypervisor. Other features, such as transparent shared page detection [118, 124], are disabled because of their significant security risks in a multi-tenant cloud [86]. Cloud providers may use live migration to update host software [129], but this has significant performance impact and hot patching is often preferred [9, 79]. We will discuss this further in §3.4.

**Bare-metal clouds.** Although the bulk of IaaS cloud workloads run in virtual machines, there is also a sizable and growing market for bare-metal cloud servers that offer dedicated machines at a premium price. The three primary reasons for a customer to choose a bare-metal instance over a VM are: (a) to avoid the CPU overhead (“virtualization tax”) for memory-intensive workloads (described in §2.1), (b) a need for predictable performance without any possible contention from other co-located VMs (or “noisy neighbors”), or (c) security/compliance concerns arising from a shared hypervisor [92]. Customers that need to run their own hypervisor may also choose bare-metal servers to avoid the substantial performance overhead of nested virtualization [20, 70]. Core slicing seeks to offer similar features at flexible granularity and consequently lower cost.

## 2.5 Summary

We see that the bulk of IaaS VMs deployed in public clouds today run with a fixed allocation of memory and discrete cores, with I/O that is or soon will be fully offloaded, and have little to no use for features of virtual machines except for isolation. At the same time, CPU vendors are adding substantial complexity (not to mention, security risks and performance overhead) to their designs to support confidential VMs. We ask the question: *since the resources assigned to cloud VMs are effectively static slices of a machine, rather than relying on complex software to check the actions of a hypervisor that is not expected to do anything at VM runtime, why not enforce those partitions in hardware?*

## 3 Design

We describe the design of core slicing, starting with its overall architecture and threat model, before detailing our proposed hardware mechanisms, and how those mechanisms can be used by system software under the control of the host to securely partition hardware among untrusted guests.

Our design goals are as follows:

1. Partition shared hardware at natural boundaries, such

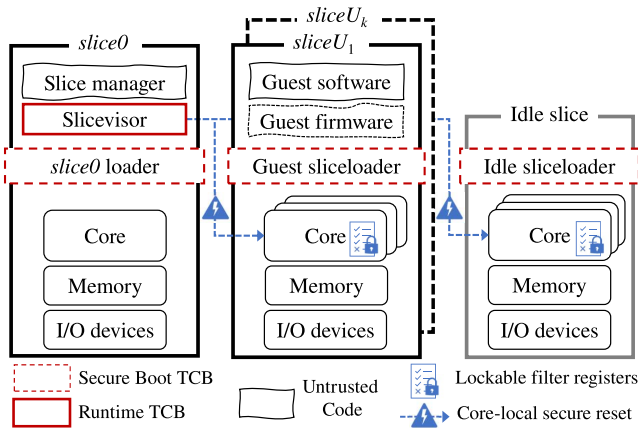


Figure 1: Core slicing system architecture.

as whole cores, while relying only on simple, easy-to-implement hardware mechanisms to ensure isolation.

2. Keep the trusted computing base small and simple, to permit a formally-verified implementation.
3. Support memory encryption and remote attestation features equivalent to confidential VMs.

### 3.1 Overview and terminology

Rather than VMs, we partition a machine into multiple guest *slices*, as shown in Figure 1 (denoted as *sliceU*). Each slice consists of a distinct set of named hardware resources: cores, memory ranges, and (virtual) I/O devices. Those resources are allocated exclusively to a slice for the duration of its execution. In the case of cores, this means that guest code runs in the highest privilege level (e.g., hypervisor mode), and controls every CPU cycle executed on that core until the slice is terminated. Like VMs, slices may start or stop at any time, and resources that were allocated together in one slice may later be partitioned among distinct slices after the first slice terminates. The key invariant is: *at any given time no two slices may share access to the same resource*. This avoids the need for any hypervisor-level mechanisms to share resources between guests, such as time-slicing VMs on a shared core or demand-paging overcommitted memory.

To allocate resources and manage the lifetime of guest slices, we rely on a distinguished control slice (termed *slice0*) running a *slice manager*. Somewhat like the host domain or root partition of a hypervisor, the slice manager is started at boot, and is responsible for creating and destroying user slices and determining their resource allocations. The slice manager runs on a core dedicated to that purpose, ideally a low-power management processor, potentially even on a separate chip as in Amazon’s Nitro system [7]. We do *not* assume that the slice manager shares memory cache-coherently with user cores, nor that it executes the same instruction set. The slice manager

software is further divided into a small, privileged portion, the *slicevisor*, that must be trusted by both cloud guests and the host, and a larger, unprivileged portion, that need not be trusted by guests. The unprivileged slice manager cannot interfere in the execution of a guest slice except for terminating it and resetting its cores. The slice manager maintains an *idle slice* to account for any unused resources. Cores in the idle slice do not execute, and merely wait to be assigned to a user slice.

To isolate resources, we rely on a new hardware mechanism, *lockable filter registers*, that restricts access to resources from a given core. Once configured and locked, these registers are read-only until the core is reset via another new mechanism, *core-local secure reset*. We require that it be restricted so that only the slicevisor can initiate a reset. A trusted loader, the *slicelocator*, is the first code to execute after a reset.

### 3.2 Security properties and threat model

Core slicing offers strong security, eliminating interference between all slices (including the control slice). Specifically:

1. The resources assigned to a slice are static from its creation until its termination.
2. A slice cannot access memory outside the slice, neither from cores nor via DMA.
3. A slice cannot interrupt cores outside the slice.
4. A slice cannot access I/O devices outside the slice.
5. Only *slice0* may terminate another slice or reset its cores.

The threat model for core slicing is comparable to other forms of trusted computing including confidential VMs and enclaves [29, 35, 64]. The host (cloud provider) and guests (cloud users) are mutually distrusting, with one caveat: a guest relies on the host to provide agreed resources (thus, denial of service is out of scope), but can check at runtime that sufficient resources (e.g., as many hardware cores as expected) are available. The cloud provider trusts the management stack executing in the control slice, which determines both which specific resources (CPU cores, memory, etc.) a guest is permitted to use, and for how long it executes.

Guests must trust only hardware, the slicevisor, and the slicelocator. The slicevisor ensures that resource allocations are disjoint (for example, that none of the memory allocated to a guest slice is ever shared with another slice) and configures hardware protection mechanisms accordingly. The slicevisor is also responsible for attesting guest slices, and forms part of the attestation root of trust. The slicelocator ensures that lockable filter registers are properly configured and locked before transitioning control to guest code on each core of a slice.

Core slicing provides substantially stronger protection from side-channel attacks than confidential VMs. Because slicing partitions a machine at core granularity, the only side channels possible are those that can be observed from another core; notably, side-channel leaks to sibling hardware threads or to a malicious hypervisor are impossible.

Hardware security features including memory encryption and cache partitioning are orthogonal to our design; if present, they allow us to strengthen the defenses against physical and cross-core side-channel attacks respectively.

### 3.3 Hardware support for core slicing

Recall that our goal is to partition shared hardware at core boundaries while relying only on simple hardware mechanisms to ensure isolation. To keep this hardware simple, a key choice we make is to *expose the underlying physical resources directly to guest software*. Specifically, a guest slice may not have access to all cores or physical memory of a machine, but the identifiers it uses (i.e., the processor IDs and physical addresses) for the resources to which it does have access are always those of the underlying hardware. The only role of our hardware extensions is to restrict the set of accessible resources on a per-core (and thus, per-slice) basis.

This design choice avoids the need for any additional translation layers (as in virtualization) but it does place some requirements on guest software (i.e., OSes or hypervisors). Specifically, guests must be able to run with non-contiguous processor IDs, and cannot assume that physical memory starts at any particular address (such as zero). Luckily, modern OSes already meet this requirement: the initial OS boot image is a position-independent binary that uses a well-specified data structure (either ACPI tables [109] or a devicetree blob [71]) to locate all accessible resources, including memory ranges and additional processors. As long as this boot-time data structure accurately describes the resources accessible to a slice, a correct guest will make no attempt to access other hardware, and it is sufficient to treat any illegal accesses as fatal.

We therefore require a hardware mechanism that can restrict access to named physical resources (physical addresses for memory and I/O, and processor IDs for inter-processor interrupts). However, by design, guest software running in a slice should be able to use the highest privilege level on those cores, which raises a conundrum: how can we configure these restrictions without a slice being able to change them?

Our solution borrows from a pattern seen in hardware support for trusted execution: we introduce *lockable filter registers* that restrict the accessible resources by *all* software (including the most privileged) running on a given core. Once configured and locked, these registers are read-only until a subsequent *core-local secure reset* regains control of the core from the slice. Similar lockable registers (sometimes referred to as “latches”) have been used to implement hardware security mechanisms such as secure boot and attestation for an

entire system [127]. To our knowledge, we are the first to propose such a technique at the granularity of a single core.

At a hardware level, our proposed secure reset mechanism is just a subset of the existing system-wide reset functionality, exposed separately at core granularity: it stops execution, resets the entire core to a well-defined architectural state (resetting locked registers), and causes the core to begin execution at a fixed address. The unique constraints we place on this mechanism are that (a) only the slicevisor running in the control slice’s privileged mode can initiate such a reset, and (b) the address of the jump target that receives control after reset remains inaccessible to any user slice. Here, we host the *slice-loader*, a small piece of trusted code similar to a secure bootloader that will reassign the core by programming and locking its filter registers before transferring control to untrusted user code, or taking it offline as part of the idle slice.

We next describe how hardware filters enable slicing of each distinct resource (memory, interrupts, cache, and I/O).

**Memory** To prevent a core from accessing any memory outside its slice, we rely on *lockable memory range registers* that restrict physical memory accesses by a core. Although we leave the precise semantics of these registers, such as the number of ranges and any alignment/size constraints, up to hardware designers, we assume that they can be configured in such a way as to restrict access to at least one contiguous range of RAM for a slice, as well as any virtual I/O devices (e.g., network and storage controllers) assigned to the slice, and any other memory-mapped registers (such as a local interrupt controller or timer) that are necessary. The minimum number of range registers is thus platform-specific, but we anticipate that around 10 ranges per core will generally suffice. More ranges will permit the slice manager greater flexibility in memory allocation, especially on multi-socket systems with non-uniform memory, but comes with a small but non-zero cost in additional hardware resources.

Range checks are trivially parallelizable and can be applied before installing a page translation in the TLB, thus having negligible runtime overhead. By contrast, virtualization relies on a nested page table which not only requires additional memory, it also increases page translation overhead and TLB pressure [3, 37, 88, 108]. However, because a slice is restricted to discrete physical ranges, its memory cannot be allocated in arbitrary pages, but instead must occupy a limited number of *contiguous* physical regions. We will evaluate the impact of this constraint on memory fragmentation in §6.4.

**Interrupts** We require hardware support to prevent a slice from sending inter-processor interrupts (IPIs) to cores outside its slice. Luckily, the address space of processor identifiers is substantially smaller than memory addresses, so we propose to use a *lockable IPI destination mask register* in preference to range checks. This register, which may in reality consist of a number of consecutive model-specific registers (e.g., four

64-bit registers for an 8-bit processor ID), permits a slice to run on any combination of cores. Of course, most workloads will benefit from adjacent cores (and caches).

**Cache** By design, any state internal to a core including the L1 cache is never shared across slices. However, shared L2 or L3 caches may raise performance interference and security concerns around cache-based side channels. To mitigate these, core slicing can make use of existing hardware support for cache partitioning [84], as long as the relevant configuration registers can also be locked or otherwise restricted.

**I/O devices** As described in §2.1–2.4, SR-IOV has been deployed by cloud providers to allow VMs to directly access networking, storage, GPUs, and more at no overhead. Just as the virtualization host OS assigns virtual functions to VMs, *slice0* configures and assigns virtual functions to slices.

I/O devices interact with software in three ways: memory-mapped registers, direct memory access, and interrupts; all three require a suitable access control mechanism. Access to memory-mapped I/O is restricted by the same range checks as regular memory, and we do not discuss it further here. However, we need a way to prevent a slice from initiating DMA transfers to any memory outside its slice. Such restrictions are typically implemented by an IOMMU [12, 49], and typical IOMMU functionality suffices for core slicing, as long as the IOMMU remains under the control of the slicevisor.

One simple approach is to configure the IOMMU to map accessible slice memory 1:1 for each virtual function belonging to a slice. Its main downside is security: all memory assigned to a slice is available for DMA. This does *not* compromise slice isolation, but, like a bare-metal system without an IOMMU, it may allow a buggy device driver to access the wrong guest memory. Rather than reprogramming the IOMMU at runtime to restrict DMA, recent work found that it is more efficient to simply allocate all I/O buffers from a dedicated pool of physical memory that remains mapped in the IOMMU [74]; this is naturally supported by core slicing, and avoids the need for runtime interaction with slicevisor to reprogram a slice’s IOMMU translations.

Besides DMA, an I/O device also sends interrupts. The interrupts of virtual functions are mapped and routed to host cores by the IOMMU, and the same mechanisms apply directly to core slicing. Since slice cores are statically assigned and there is no host hypervisor, direct interrupt mapping is substantially simpler than VMs [40].

One unique challenge of SR-IOV is that virtual functions do not implement normal PCI configuration space registers. Rather, a hypervisor typically emulates configuration space accesses for an assigned virtual function. For slices, we could rely either on an enlightened guest to avoid the need for such virtualization (as in our x86 prototype, see §5), or else on a custom PCI “card” [6, 36] to emulate a standard memory-mapped configuration space within its own device window.

### 3.4 Slice management

We now turn our attention to the *slice0* software stack responsible for resource allocation, slice lifetime, and a handful of runtime services. This may run on a dedicated host core, a low-power management core, or a separate SoC. We require only that it (a) has hardware privilege separation, (b) is able to trigger secure resets of guest cores, and (c) shares *some* memory, not necessarily cache-coherently, with those cores.

Recall that only the privileged portion of the slice manager, the slicevisor, is trusted by guest slices. The unprivileged slice manager has no access to guest slice memory or cores. The slicevisor implements all security-sensitive aspects of the process of creating, starting, stopping, and deleting a slice. Its primary role is to ensure that no resource is ever accessible to two slices at the same time; this includes checking that memory ranges assigned to slices are disjoint, and ensuring that all cores of an expired slice have been stopped via a secure reset prior to reassigning any resources of that slice.

To keep the slicevisor as simple as possible (and permit its eventual implementation in formally-verified code), it does not directly allocate resources, but merely checks the correctness of resource assignments provided by the unprivileged slice manager. This permits the unprivileged slice manager to implement flexible policies to choose the memory ranges and cores assigned to slices, without the need to trust them.

The only other code that must be trusted by guests is the *sliceloder* which is the first thing to run after a core is reset by the slicevisor. It is responsible for configuring and locking the per-core access filters, and then coordinating with other loaders of the slice to securely boot the guest OS. To do this, it relies on a *slice table* of configuration information maintained by the slicevisor (and inaccessible to untrusted software).

**Starting a guest** To create a slice, the unprivileged slice manager assigns available cores and memory, and determines the configuration of virtual I/O devices. Next, it invokes the slicevisor to create the slice, which rejects the request if the new slice includes any resource shared with another slice (including *slice0*), other than the idle slice. Otherwise, the slicevisor updates the slice table, programs devices to configure virtual functions, and updates IOMMU translations as described earlier. At this stage, the assigned cores remain idle.

To start the new slice, the slicevisor resets the relevant cores, which causes them to execute the *sliceloder* and follow a secure boot flow. After reading the relevant configuration from the slice table (recall that at this point the *sliceloder* is trusted and has unrestricted access to system memory), the loader determines whether the current core belongs to the idle slice or a new guest slice. It then programs and locks the core’s access filters for memory ranges and IPs, before synchronizing with the loaders (if any) for other cores in the slice. The rest of the boot process has no access to memory or cores of other slices. It zero-fills slice memory ranges, before



copying the guest's boot image and transferring control.

**Terminating a guest** Unless they are unresponsive, guest OSes will generally execute a controlled shutdown initiated via an out-of-band signal. To finally terminate a slice, the slice manager invokes the slicevisor which clears the relevant configuration in the slice table. Then, to stop the guest cores (which no longer have access to the slice table), the slicevisor reassigns them to the idle slice and resets them.

**Auxiliary services** At runtime, the slice manager exposes a simple shared memory device (much like a virtual I/O device) to guest slices, permitting an enlightened guest OS driver to initiate a shutdown or reset of the slice, or access slow-path emulated I/O devices (such as a virtual serial port and console) for which no offload device is warranted.

**Unsupported features** The functionality of a guest slice is similar to bare-metal clouds, and lacks advanced VM features such as live migration. This does not preclude a guest from implementing its own mechanisms (e.g., by running its own hypervisor, or doing so at process level [34]), but it does prevent a cloud provider from transparently migrating guests to implement software upgrades [129]. Since core slicing eschews the use of a hypervisor and runs the entire host stack on a dedicated management core, we do not anticipate that updates will require live migration. In particular, we expect that it will be possible to update the slice manager and slicevisor without any guest interruptions.

### 3.5 Attestation and memory encryption

We assume that hardware implements a root of trust for secure boot, permitting the initial bootloader and slicevisor to be cryptographically measured and attested. In turn, the slicevisor attests individual slices; this includes a measurement of slice configuration, the sliceloader code, and the guest image. Once booted, a guest can prove to a remote verifier that its slice is configured as expected and that its isolation is enforced by trusted slicevisor and sliceloader implementations.

As described earlier, slice memory is strongly isolated to defend against software attacks. To defend against physical attacks on memory, such as cold-boot and memory-bus attacks [43, 89], core slicing can leverage memory encryption hardware. The details of memory encryption are orthogonal to our design, and we expect to leverage existing platform mechanisms. In particular, because only trusted components (sliceloader, guest code, and I/O devices within DMA regions) can access slice memory, it is irrelevant to slice guests whether memory is encrypted by a random system-wide key (as in Intel SGX and Arm CCA), one of a set of random keys (as in Intel TDX), or a unique key for each guest (as in AMD SEV).

If per-slice memory encryption is nevertheless desired, we assume that the hardware will provide a suitable interface for

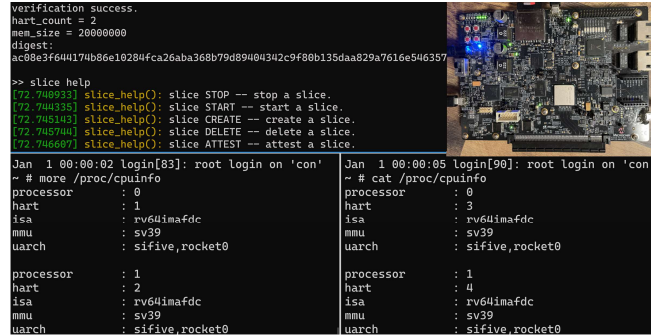


Figure 2: RISC-V prototype with the slice manager (upper left) and two Linux guest slices (lower terminals).

the slicevisor to configure a unique slice memory encryption key. In that case, when initializing a new slice, the sliceloader starts out in an unencrypted context before enabling encryption to load and boot the guest. Similarly to other confidential computing architectures [15, 54], once the guest boots, attested I/O devices may access the encrypted memory using the guest's encryption context.

## 4 RISC-V Prototype

We demonstrate the feasibility of our design with a RISC-V prototype. We chose RISC-V because an existing feature closely approximates the lockable filter registers required by core slicing. Our prototype runs in two environments: on a modified version of QEMU, and on an unmodified Microchip PolarFire Icicle board with a SiFive FU540-C000 SoC shown in Figure 2, the latter with some limitations due to partial support for our requirements. As the guest OS we run Linux.

Common RISC-V CPUs implement three privilege levels: machine (M) mode for firmware, supervisor (S) mode for an OS kernel, and user (U) mode for applications. Our SiFive SoC includes four general-purpose application cores and one less-powerful monitor core that implements a limited instruction set with only M and U modes. The monitor core is ideal for *slice0*, with the slicevisor running in privileged M-mode, and the rest of the slice manager in user mode. The remaining four application cores are available for guest slices in arbitrary combinations (i.e., up to four single-core guests).

**Memory** In addition to typical address translation mechanisms, RISC-V supports physical memory protection registers [93, §3.6] which can be programmed to restrict access to physical address ranges on a per-core basis. These registers are grouped into 8 or 16 PMP entries. Each PMP entry consists of a configuration register and an address register that specify the access permission (read/write/execute) to a particular region. Once programmed, PMP checks apply to all memory accesses from user and supervisor modes, and

Table 1: PMP permissions by physical region.

| Physical address range                     | <i>slice0</i> M-mode | <i>slice0</i> U-mode | <i>sliceU<sub>1</sub></i> |
|--------------------------------------------|----------------------|----------------------|---------------------------|
| <i>sliceU<sub>1</sub></i> RAM              | —                    | —                    | RWX                       |
| <i>slice0</i> trusted RAM                  | RWX                  | —                    | —                         |
| <i>slice0</i> untrusted RAM                | RWX                  | RWX                  | —                         |
| <i>sliceU<sub>1</sub></i> bus              | RW                   | —                    | RW                        |
| Other <i>sliceU</i> bus                    | RW                   | —                    | —                         |
| Cache controller                           | RW                   | —                    | —                         |
| Reset unit                                 | RW                   | —                    | —                         |
| Interrupts to <i>sliceU<sub>1</sub></i>    | —                    | —                    | RW                        |
| Interrupts to <i>slice0</i>                | RW                   | —                    | RW                        |
| Physical I/O devices*                      | RW                   | —                    | —                         |
| <i>sliceU<sub>1</sub></i> virtual devices* | —                    | —                    | RW                        |

\* Not implemented due to hardware limitations.

optionally from machine mode. Finally, as required by core slicing, PMP registers include a *lock* bit that, once set, prevents any subsequent modifications until a reset. Our prototype sliceloder configures and locks the PMP entries for each core in the slice before booting the OS. Thus, code in a slice cannot access physical addresses outside its slice, even from the most privileged machine mode.

Besides memory, other locked PMPs grant access to a slice’s hardware resources (described below), and enforce privilege separation between the slice manager and slicevisor on the monitor core. Table 1 summarizes their configuration.

**Interrupts** Recall that our design calls for lockable mask registers restricting the destinations for inter-processor interrupts. We found that although RISC-V lacks such a feature, the careful use of PMPs permits an equivalent mechanism. To send an IPI on the SiFive SoC, the source core writes to a memory-mapped register of the destination’s “core-local interruptor” [102]. Because the register’s address is unique for every destination, we can use the source core’s PMPs to restrict the addressable (and thus interruptible) destination cores. Access to core-local timers is restricted similarly.

**I/O devices** Our board includes several I/O controllers, and our prototype grants access to slices using PMPs and routes interrupts accordingly. We were unable to prototype SR-IOV support due to a lack of suitable hardware, such as an IOMMU. The RISC-V community has proposed PMP-like mechanisms to restrict DMA [103], but these are not yet available.

**Cache partitioning** The SiFive SoC includes a 2 MiB shared L2 cache that supports *way masking*, allowing each cache master to be restricted to a subset of the 16 total ways. Since each core’s L1 I- and D-cache act as separate masters for the L2 cache, we can flexibly partition it by enabling distinct way sets for each slice. Our implementation scales the size of a slice’s cache partition with the number of cores in that slice (i.e., four ways per core); thus, larger slices enjoy a

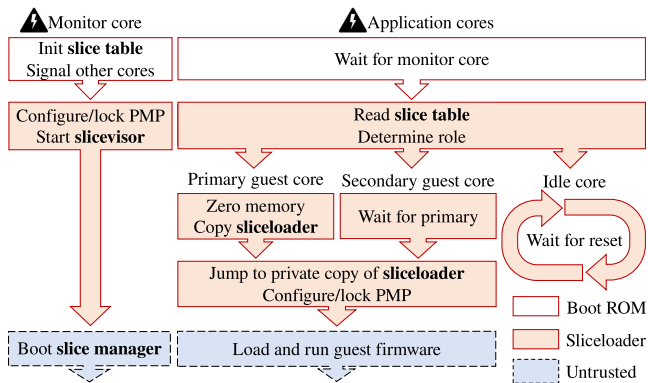


Figure 3: Boot flow for each core after reset.

larger share of the cache. Untrusted code cannot change the cache configuration because the cache control registers are enabled only in the monitor core’s M-mode PMP registers.

**Core reset** Recall that our design relies on a secure core-local reset to re-establish trusted control of a core from a user slice. Unfortunately, our board does not expose per-core reset signals, requiring a full system reset that reboots the entire board to clear any PMP lock bits. We have taken a number of approaches, with different trade-offs, to avoid this limitation.

First, we modified QEMU to implement a new device that exposes per-core reset registers; our prototype slicevisor uses these to reassign cores.

Second, we modified the slicevisor to create slices at boot time using a pre-determined configuration. On our board, such slices cannot be destroyed without a whole-system reset because their PMPs are locked. However, despite the loss of flexibility, this provides a strong security guarantee.

Finally, to test our ability to destroy and create slices on hardware, we implemented an insecure slice manager with an alternative software-based reset mechanism. In this version, PMP lock bits are not set. This provides no meaningful security (a malicious slice could reconfigure PMPs), but permits a cooperative slice to emulate a “secure” reset upon receipt of an IPI by clearing PMPs and jumping to the sliceloder.

**Sliceloder and guest firmware** As shown in Figure 3, the actions of the sliceloder vary depending on the slice to which the running core is assigned. When booting a guest, the sliceloder copies itself to private slice memory before setting PMPs, because doing so makes the main copy inaccessible.

When Linux boots, it infers the system configuration from a devicetree blob [71] provided by the firmware. This describes the available memory, cores, and devices on a given platform. To allow Linux to boot in a slice, we enlightened the OpenSBI bootloader [94] to run as untrusted code inside a newly-created slice. It constructs a devicetree describing the slice configuration before booting Linux. Given an appropri-

ate devicetree, Linux itself required no modifications to run with arbitrary physical memory ranges or core IDs.

**Slice communication** To enable communication between the slice manager and guests, we implemented a *slice bus* message transport, using a region of shared memory between each guest slice and *slice0*, with IPIs for signaling.

**Attestation** To prove that a guest slice runs only the image expected on a trusted platform, we implemented *measured boot* in the sliceloader and *attestation* in the slicevisor. When creating a slice, the sliceloader measures guest code and stores its hash in *slice0* trusted memory. To attest, slicevisor generates an attestation report including the guest measurement and user-provided data, signed by a slicevisor-held key derived from a hardware root of trust. A remote party can verify the attestation using the device public key and the TCB report.

**Limitations** Due to hardware limitations, our prototype lacks support for memory encryption. We do not yet implement virtual serial ports, but assign a UART to each slice.

## 5 x86 Prototype

We also implemented an x86 prototype. This lacks security isolation, but permits us to experiment with SR-IOV devices and compare performance to the state-of-the-art in hardware virtualization. We discuss ideas for actual hardware support on x86 platforms later, in §7.

Since there is no security, we simplified our management stack by building on top of Linux. Specifically, we run Linux on the bootstrap core, using kernel parameters that restrict Linux to a single core and a minimal amount of physical memory. Non-boot cores remain idle, in the *wait-for-startup-IPI* state. A privileged process is later responsible for configuring and booting slices with a user-specified set of cores, range of physical memory, and set of PCI devices.

**Booting a slice** To boot a slice, we load the guest kernel into the chosen physical memory range (accessed via `/dev/mem`), construct ACPI and E820 tables describing resources available to the slice, and then send a startup IPI to the slice’s first core. This runs a tiny (48-instruction) real-mode bootloader that constructs a page table, switches to 64-bit mode, and enters the slice kernel, which then boots as usual, sending further startup IPIs to other cores.

**Guest enlightenments** Because all the host hardware remains accessible, the Linux guest needed a few modifications. We disabled the `CONFIG_DMI` and `CONFIG_X86_MPPARSE` build options to prevent the slice kernel discovering these legacy firmware tables (and hardware they

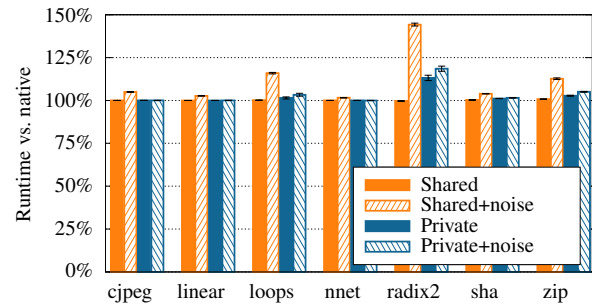


Figure 4: RISC-V CoreMark results (lower is better).

describe) in the system BIOS. We fixed a bug that unconditionally enabled interrupts from the legacy PIC despite ACPI flagging it as absent. Finally, we added 283 lines of code to (a) use an SR-IOV virtual function without a virtual configuration space, and (b) enable only a subset of PCI devices. These enlightenments would be irrelevant to a hardware implementation. In particular, host firmware and devices would be inaccessible to slices, and (as described in §3.3) an I/O device could emulate standard PCI configuration space.

## 6 Evaluation

This evaluation seeks to answer the following questions:

- What is the performance overhead of core slicing? (§6.1)
- How does the design of core slicing translate into concrete security benefits for guest slices? (§6.2)
- What is core slicing’s hardware complexity? (§6.3)
- Does the need for contiguous physical memory lead to slice allocation failures due to fragmentation? (§6.4)

### 6.1 Performance

Our experiments run on the RISC-V board described in §4 with a 16-way 2 MiB L2 cache and 1 GiB of DRAM, and on an HP Z8 workstation with two Intel Xeon 4214 12-core CPUs (HyperThreading disabled), 64 GiB of RAM, a Mellanox ConnectX-4 25GbE NIC, and a Samsung PM1735 NVMe SSD. Another HP Z8 with two Xeon 4108 CPUs and the same NIC serves as a client.

**RISC-V** We run CoreMark PRO [33] and focus on two questions: (a) does a slice achieve bare-metal performance, and (b) what is the impact of a “noisy neighbor” slice?

For the native baseline, we enable two application cores and 512 MiB memory. This allows a fair comparison with the slice measurements, in which we launch two guests, *sliceU<sub>1</sub>* and *sliceU<sub>2</sub>*, each with two cores and 512 MiB memory. As shown by the *shared* results, performance in a slice exactly matches

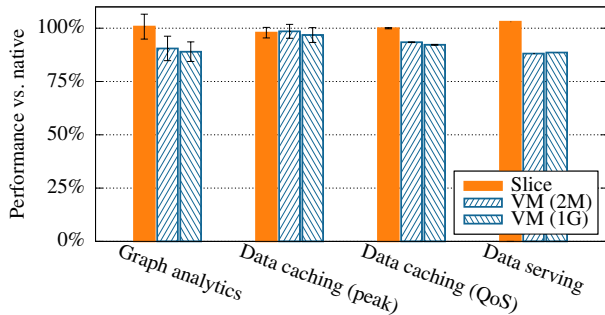


Figure 5: x86 CloudSuite results (higher is better).

bare-metal execution. To determine the impact of cache contention, we run a workload in *sliceU<sub>2</sub>* as a noisy neighbor (denoted by ‘noise’) that repeatedly accesses a 2 MiB array (the same size as the L2 cache). Unsurprisingly, the average runtimes (denoted by *shared+noise*) increase depending on the given workload’s cache intensity. Specifically, *radix2* suffers the most (almost 50% overhead) since it operates on a 512 KiB array and suffers frequent cache misses. As demonstrated by *private+noise*, when the cache is partitioned, a noisy neighbor has a substantially smaller impact, with all workloads achieving stable performance, and outperforming the *shared+noise* configuration. The runtimes in *private+noise* increase slightly (< 5%) with noise compared to *private*, due to contention for the memory controller.

**x86** We compare slice performance to equivalent VMs using CloudSuite 3.0 [87]. Both slice and VM guests have 8 cores and 16 GiB of RAM (allocated on the same NUMA node), with NIC and NVMe virtual functions for I/O. We run VMs under KVM using full hardware acceleration including virtual APICs, and confirm via performance counters that the only significant source of VM exits is to service and emulate timer interrupts. To minimize memory management overhead we use pre-allocated and locked huge-pages (2 MiB and 1 GiB).

The results in Figure 5 are scaled to a native baseline with the same cores/memory. *Graph analytics*, which uses Apache Spark and GraphX to run PageRank on a large Twitter dataset, runs 10% slower in a VM. *Data caching* models a Twitter cache server with Memcached. Despite sustaining a similar peak throughput, the VMs have higher jitter and thus perform up to 8% worse while meeting the published QoS target of 10 ms p95 latency. This appears to be due to the extra TLB pressure of nested paging: DTLB misses are more than 3× native for 1 GiB pages, and 5× for 2 MiB pages. Finally, *Data serving* runs Apache Cassandra with 10M records of YCSB workload A and incurs a 12% throughput penalty in a VM.

**Summary** Core slicing achieves bare-metal performance without the overhead of virtualization, which remains significant for memory-intensive workloads, even with huge pages.

Table 2: Size of software TCB.

|             | Source lines <sup>a</sup> | Executable code <sup>b</sup> |
|-------------|---------------------------|------------------------------|
| slicevisor  | 3,609                     | 18 KiB                       |
| sliceloader | 3,607                     | 11 KiB                       |
| Total       | 4,826 <sup>c</sup>        | 29 KiB                       |

<sup>a</sup> Non-header lines, counted by `clloc` [30]. <sup>b</sup> Size of text section, from `binutils size`. <sup>c</sup> A library common to both is counted once.

Furthermore, hardware cache partitioning (as on RISC-V) not only lessens the impact of a noisy neighbor, it can also eliminate both cache contention and cache-based side channels.

## 6.2 Security

**Size of trusted computing base** Although no guarantee of security, a small TCB helps make formal verification tractable. Table 2 reports the executable source and binary sizes of our prototype. These are comparable to the firmware for Arm CCA (4.3 kLOC [69]), although we note that core slicing’s TCB eschews runtime interaction with a running guest, and thus its attack surface is drastically simpler.

**Side channels** By partitioning a machine at core granularity and without a trusted hypervisor, core slicing avoids either the host or another guest running concurrently on a core. This is inherently more secure than either reducing the hypervisor’s size [99, 100] or de-privileging it [13, 17, 53]. We also gain a systematic defense from a wide variety of CPU side-channel attacks. Following our threat model (§3.2), we consider two classes of attacker: guest attackers who may run arbitrary code in a guest slice, and the host attacker who controls untrusted code in *slice0*. We describe the extent to which these may compromise a guest’s confidentiality.

**Cache side-channel attacks** are defeated by cache partitioning and memory isolation. Because slices never share memory, a guest attacker cannot steal secrets by analyzing whether a co-located tenant accesses shared memory addresses. Because caches are partitioned, the attacker cannot observe how many cache lines are used by the guest per cache set. *slice0* is similarly restricted. Thus, given suitable hardware support, core slicing eliminates cache-based side channels. It also defeats all side-channel attacks where the attacker executes on the victim core, including sibling threads [4, 98].

**Transient execution attacks** can leak secrets through the side effects of speculative memory accesses [62, 72], and can break isolation between hypervisors and VMs. Core slicing relies on lockable filter registers to restrict memory access. Because filters only perform a range comparison (with no memory-bound table walk), they derive no significant benefits from speculation; e.g., current RISC-V CPUs cache PMP range checks in the TLB along with address translations [83].



Cross-core transient execution leaks were recently observed on Intel CPUs [91]; although these remain a threat, we expect them to be drastically simpler for vendors to identify and fix in future CPUs, since few instructions access uncore state. Of note, the demonstrated attack relies on delaying a victim enclave’s execution with page faults and exceptions, which is impossible across a slice boundary.

**Page-fault-** and **page-table-based attacks** are highly exploitable on TEEs where an untrusted hypervisor manages guest memory using a nested page table. In these attacks, a host learns the guest’s secret-dependent memory access pattern via page faults, page table access/dirty bits, or even cache contention with hardware table walks. Core slicing prevents this by allocating physical memory directly to guests. In addition, memory encryption can suffer from **ciphertext-only attacks** (e.g., dictionary attacks) with weak encryption algorithms. In our design, lockable filter registers prevent access to guest memory, including ciphertext, even by the host.

Side channels in resources shared by multiple cores remain, including power [61], row-hammer [59], cold-boot [43], and memory bus [89] attacks. Those can be mitigated with additional orthogonal hardware support.

### 6.3 Hardware complexity

To estimate the hardware cost of supporting core slicing, we extended the default *tiny* configuration of the RISC-V Rocket Chip implementation [18]. To measure the overhead of adding lockable filter registers, we doubled the number of PMPs from 16 to 32 (as might be necessary when existing PMPs are required for other uses), resulting in only a 3% increase in total FPGA resources. We also added per-core resets and a reset device for the monitor core, for 1.7% extra resources. Since these results are for an embedded core, we expect the fraction of resources required on a server-grade CPU to be much smaller.

### 6.4 Impact of physical contiguity

Unlike VMs, slices require *contiguous* physical memory, and the memory assigned to a slice cannot be changed without terminating and restarting it. Thus, it is possible that the available memory on a node becomes fragmented over time, leading to a situation where sufficient free memory exists to support a new slice, but cannot be used as it is not contiguous. Whether this is a problem in practice depends on both the pattern of memory allocations (i.e., the order of slices created and destroyed) and the policy implemented by the memory allocator (i.e., which region of memory to allocate for any given request). In this section, we report the analysis of VM start/stop events on a public cloud workload, modeling the effects of memory allocation policy and hardware capabilities.

The trace we use is similar to the VM allocation trace of Hadary et al. [42]. It includes all VM start and stop events (in

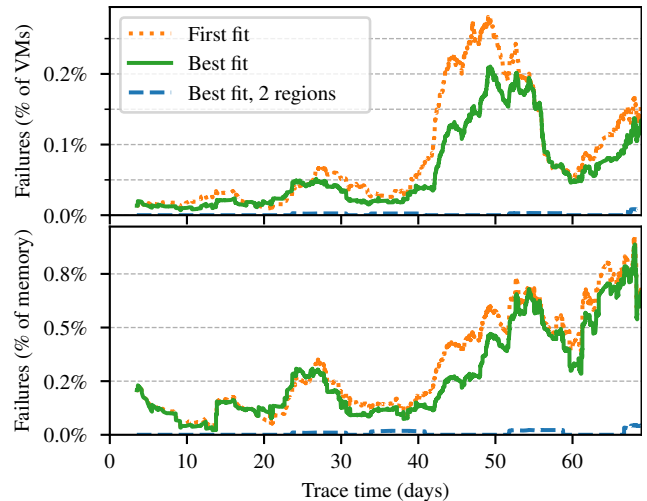


Figure 6: Rate of slice allocation failures due to memory fragmentation over a 7-day moving window.

excess of 750k events) for an Azure cluster, and was gathered over two months in mid-2021. Each VM has a type [81] that defines its resource allocation, including its memory size (other resources are irrelevant to our analysis).

In this analysis we ask the question: *how often does fragmentation prevent the allocation of a slice on a node when a VM would have succeeded?* Thus, although it may be beneficial for the cloud scheduler to use node-level memory contiguity in its placement decisions (allocating slices on nodes to reduce fragmentation), we leave the mapping of VMs to nodes unchanged and model slices as a drop-in replacement. Since our focus is memory allocation, we model every VM as a slice with physically contiguous memory. In reality, we expect that some types (such as burstable VMs) would continue to run as VMs on the cloud provider’s hypervisor either within a slice of a larger machine, or using dedicated machines.

The results of our analysis are shown in Figure 6. We experimented with a variety of memory allocators, and found unsurprisingly that a best-fit policy (which places each new slice in the smallest free region of sufficient size) minimized fragmentation and thus allocation failures. Other policies, including a traditional buddy allocator, performed uniformly worse and are not shown on the figure. Calculating the best allocation will take slightly longer, but given the low rate of slice instantiations relative to traditional memory allocation workloads, this overhead is not expected to be significant.

As shown in the figure, the allocation failure rate for fully contiguous memory is less than 0.3% of VMs in the trace, representing less than 1% of the total requested memory (since larger VMs are more likely to fail allocation). We also modeled the effect of permitting multiple contiguous regions per slice. With hardware support for two memory regions per slice, the memory allocator is able to split large slices across two distinct allocations, and the failure rate drops further to

less than 0.1% (only 6 failures across the entire trace). With three regions per slice, there are no failures. Overall, we conclude that restricting slices to contiguous memory does not pose a significant constraint for cloud operators. A different analysis by Teabe et al. [108] reached the same conclusion.

## 7 Discussion: core slicing beyond RISC-V

For a prototype implementation of core slicing, RISC-V had several advantages; most significantly, the use of physical memory protection registers allowed us to implement the bulk of our design on unmodified hardware. However, our design does not depend on RISC-V, and we ultimately hope to see it adopted by the x86 and Arm architectures that dominate today's cloud. This section discusses some of the challenges in doing so, and offers guidance to hardware designers.

The obvious first step in adapting an existing architecture to support core slicing would be to implement our hardware requirements: lockable filter registers to restrict a core's ability to access memory and send inter-processor interrupts, and a secure core-local reset to regain control of it. Of course, the details matter, and thanks to the long evolution of these architectures, there are many interactions with existing architectural features that must be considered.

Recall that our design goal with core slicing is to give guest software unfettered bare-metal access to a single core (or set of cores). As a first rule of thumb, we propose that *resource restrictions imposed via filter registers should take priority over other core-level architectural features*. This implies that existing features granting privileged access to memory, such as hardware shadow stacks [52] or secure-world memory regions [16] must be constrained by lockable address filters.

Second, *any hardware resources that are shared by more than one core must be restricted*. This includes peripherals, memory and cache controller configurations, and power management registers, among others. In RISC-V systems, such registers are memory mapped and thus restricted by PMPs, but on x86 they are configured via model-specific registers (MSRs) that occupy a distinct address space accessible to privileged software on each core (Arm system registers are similar). The access restriction could be implemented via further filter registers, or as a simpler alternative, access to these resources could be limited to the management core running the sliceloader. For the specific case of x86 MSRs, we expect that the MSR bitmaps found in the VM control block will serve as a useful starting point in determining the appropriate policy. Finally, the x86 legacy I/O address space must be filtered or (for legacy-free guests) blocked outright.

Finally, *the platform must not depend on firmware running on guest cores*. Thus, the system design should avoid the need for platform firmware in x86 system management mode or Arm EL3 on general-purpose cores. The motivation for this requirement is the same as that of core slicing: to avoid relying on intra-core privilege separation due to its demonstrated

weakness. In our view, firmware tasks are better delegated to a dedicated management core (along with the slicevisor).

## 8 Related work

**Direct hardware assignment** We discussed secure hypervisors and confidential VMs in §2.2.

NoHype's [58, 107] central security goal is to protect a *trusted* cloud provider and its legitimate customers from rogue VMs that try to exploit vulnerabilities in the hypervisor or the associated virtualization stack. NoHype achieves this goal by removing all run-time interfaces that traditional hypervisors expose to traditional VMs. The security goals of core slicing reach significantly further and address threats that have emerged during the decade since NoHype was designed. In addition to protecting the cloud infrastructure from rogue guest VMs, core slicing protects guests from the *untrusted* cloud provider. This task is complicated by an ever-growing array of microarchitectural attacks that can leak information out of VMs. Therefore, core slicing does not allow any cloud provider code to run on a guest's processor cores. In contrast, NoHype requires a highly privileged "temporary hypervisor" on those cores.

Core slicing relies on simple lockable filter registers to confine guests which enjoy bare-metal control over cores, and may run their own hypervisor. NoHype relies on conventional processor privileges; guests thus lack access to virtualization extensions, must be modified to avoid VM exits (notably, they must not execute CPUID, including in user mode), and must ignore spurious interrupts from other guests (both a side channel and a denial-of-service attack).

TrustOSV [120] and Quest-V [123] minimize runtime guest-hypervisor interactions by statically assigning cores and memory. The core of TrustOSV is a *microhypervisor* that uses nested paging to constrain guest memory accesses. Quest-V replaces a global hypervisor by trusted per-core monitors that also run in host/root mode and use nested paging. TrustOSV reduces trust in the cloud provider by attesting the microhypervisor and exposing a limited management interface. In contrast to core slicing's use of I/O offload, TrustOSV exposes a virtual NIC which is also the basis of its storage.

**Space partitioning** The use of core-granularity spatial partitioning [47, 73] for resource and security isolation has been explored in the context of prior many-core systems including the Tiler TILE64 [122], Intel single-chip cloud computer [75] and M3 [19], and the core idea dates back at least as far as IBM's logical partitioning feature from the 1980s [21]. Core slicing builds on the same mechanism, but is unique in its adoption of the confidential computing threat model, with a clear separation of host and guest trusted computing base. This leads us to the use of a unique mechanism combining per-core secure reset with lockable filter registers to

enable attested boot while minimizing the guest TCB. Past designs, including those cited above, permit a guest's accessible resources to be reconfigured at runtime by a privileged management core, requiring substantially more trust by the guest in the host's resource manager.

**RISC-V security** Several systems use PMP hardware for different goals. Keystone [64] is a framework for trusted execution environments similar to Intel SGX [46]. The OpenSBI bootloader can partition a machine into static PMP-isolated domains at boot time [95]. MultiZone [45] isolates software components (e.g., core RTOS and communication stack).

## 9 Conclusion

VMs are the basis of cloud isolation, but relying on them for confidential computing carries a serious risk from side channels. Core slicing offers an attractive middle ground between bare-metal servers and confidential VMs. By partitioning hardware at natural boundaries (discrete cores and contiguous physical memory ranges), it enables VM-like functionality and bare-metal performance with strong isolation.

Our prototypes are available at <https://github.com/MSRSSP/core-slicing>.

## Acknowledgments

We thank Luke Marshall for help preparing the VM traces in §6.4. The detailed reviews we received from OSDI'22, ASPLOS'23 and OSDI'23 along with feedback from our shepherd Ed Bugnion helped us greatly improve the paper.

## References

- [1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2006. doi: [10.1145/1168857.1168860](https://doi.org/10.1145/1168857.1168860).
- [2] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, pages 419–434, Feb. 2020. ISBN 978-1-939133-13-7. <https://www.usenix.org/conference/nsdi20/presentation/agache>.
- [3] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-It-Yourself virtual memory translation. In *Proceedings of the 44th IEEE International Symposium on Computer Architecture*, pages 457–468, 2017. doi: [10.1145/3079856.3080209](https://doi.org/10.1145/3079856.3080209).
- [4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Taveri. Port contention for fun and profit. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 870–887, 2019. doi: [10.1109/SP.2019.00066](https://doi.org/10.1109/SP.2019.00066).
- [5] *Enhanced networking on Linux*. Amazon Web Services, Dec. 2022. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html>.
- [6] *AWS Nitro System*. Amazon Web Services, Dec. 2022. <https://aws.amazon.com/ec2/nitro>.
- [7] *The Security Design of the AWS Nitro System*. Amazon Web Services, Nov. 2022. <https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html>.
- [8] *Amazon EBS and NVMe on Linux instances*. Amazon Web Services, Dec. 2022. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nvme-ebs-volumes.html>.
- [9] *Amazon EC2 Maintenance Help Page*. Amazon Web Services, 2022. <https://aws.amazon.com/maintenance-help>.
- [10] *Amazon EC2: Burstable performance instances*. Amazon Web Services, Dec. 2022. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html>.
- [11] *Amazon EC2: Instance types*. Amazon Web Services, Dec. 2022. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html>.
- [12] *AMD I/O Virtualization Technology (IOMMU) Specification*. AMD, Dec. 2016. Publication #48882 rev. 3.00 [https://developer.amd.com/wordpress/media/2013/12/48882\\_IOMMU.pdf](https://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf).
- [13] *AMD SEV-SNP: Strengthening VM isolation with integrity protection and more*. AMD, Jan. 2020. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>.
- [14] AMD. AMD server vulnerabilities, Nov. 2021. Security Bulletin ID AMD-SB-1021 <https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1021>.
- [15] *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization*. AMD, Mar. 2023. <https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf>.



- [16] *Building a Secure System using TrustZone Technology*. ARM Limited, Apr. 2009. Ref. PRD29-GENC-009492C.
- [17] *Arm Realm Management Extension (RME) System Architecture*. Arm Limited, Nov. 2021. Document DEN0129 ver. A.b <https://developer.arm.com/documentation/den0129/ab>.
- [18] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr. 2016. <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [19] N. Asmussen, M. Völöp, B. Nöthen, H. Härtig, and G. Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 189–203, 2016. doi: [10.1145/2872362.2872371](https://doi.org/10.1145/2872362.2872371).
- [20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010. URL <https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization>.
- [21] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, Mar. 1989. ISSN 0018-8670. doi: [10.1147/sj.281.0104](https://doi.org/10.1147/sj.281.0104).
- [22] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies*, 2017. doi: [10.5555/3154768.3154779](https://doi.org/10.5555/3154768.3154779).
- [23] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original VMware Workstation. *ACM Transactions on Computer Systems*, 30(4), Nov. 2012. doi: [10.1145/2382553.2382554](https://doi.org/10.1145/2382553.2382554).
- [24] R. Buhren, C. Werling, and J.-P. Seifert. Insecure until proven updated: Analyzing AMD SEV’s remote attestation. In *Proceedings of the 26th ACM Conference on Computer and Communications Security*, pages 1087–1099, 2019. doi: [10.1145/3319535.3354216](https://doi.org/10.1145/3319535.3354216).
- [25] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, pages 142–157, 2019. doi: [10.1109/MSEC.2019.2963021](https://doi.org/10.1109/MSEC.2019.2963021).
- [26] C. Cohen, J. Forshaw, J. Horn, and M. Brand. AMD secure processor for confidential computing security review. Google Project Zero, May 2022. <https://googleprojectzero.blogspot.com/2022/05/release-of-technical-report-into-amd.html>.
- [27] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 189–202. ACM, 2011. doi: [10.1145/2043556.2043575](https://doi.org/10.1145/2043556.2043575).
- [28] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 153–167, Oct. 2017. doi: [10.1145/3132747.3132772](https://doi.org/10.1145/3132747.3132772).
- [29] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, pages 857–874, Aug. 2016. ISBN 978-1-931971-32-4. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan>.
- [30] A. Danial. cloc: Count lines of code, 2022. <https://github.com/AIDanial/cloc>.
- [31] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher. SynthCT: Towards portable constant-time code. In *Proceedings of the Annual Network and Distributed System Security Symposium*, Feb. 2022. doi: [10.14722/ndss.2022.24215](https://doi.org/10.14722/ndss.2022.24215).
- [32] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, pages 1–10. IEEE, Jan. 2010. doi: [10.1109/HPCA.2010.5416637](https://doi.org/10.1109/HPCA.2010.5416637).
- [33] *CoreMark PRO*. EEMBC, July 2019. v1.1.2743 <https://www.eembc.org/coremark-pro>.



- [34] P. Emelyanov. Checkpoint/restore in userspace (CRIU), 2022. <https://criu.org>.
- [35] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 287–305, Oct. 2017. doi: [10.1145/3132747.3132782](https://doi.org/10.1145/3132747.3132782).
- [36] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, pages 51–66, Apr. 2018. ISBN 978-1-939133-01-4. <https://www.usenix.org/conference/nsdi18/presentation/firestone>.
- [37] J. Gandhi, M. D. Hill, and M. M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *Proceedings of the 43rd IEEE International Symposium on Computer Architecture*, pages 707–718, 2016. doi: [10.1109/ISCA.2016.67](https://doi.org/10.1109/ISCA.2016.67).
- [38] *Gartner Says Worldwide IaaS Public Cloud Services Market Grew 40.7% in 2020*. Gartner, June 2021. <https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020>.
- [39] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Kostl, and D. Gruss. SQUIP: Exploiting the scheduler queue contention side channel. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, pages 468–484, 2023. doi: [10.1109/SP46215.2023.00027](https://doi.org/10.1109/SP46215.2023.00027).
- [40] A. Gordon, N. Amit, N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–422, 2012. doi: [10.1145/2150976.2151020](https://doi.org/10.1145/2150976.2151020).
- [41] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017. doi: [10.1145/3065913.3065915](https://doi.org/10.1145/3065913.3065915).
- [42] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: VM allocation service at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 845–861, Nov. 2020. ISBN 978-1-939133-19-9. <https://www.usenix.org/conference/osdi20/presentation/hadary>.
- [43] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60, July 2008. <https://www.usenix.org/conference/17th-usenix-security-symposium/lest-we-remember-cold-boot-attacks-encryption-keys>.
- [44] F. Hetzelt and R. Bühren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52 (7):129–142, 2017. doi: [10.1145/3050748.3050763](https://doi.org/10.1145/3050748.3050763).
- [45] *MultiZone Security Reference Manual*. HEX-Five, Sept. 2020. <https://github.com/hex-five/multizone-sdk/raw/8c92f55/manual.pdf>.
- [46] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. doi: [10.1145/2487726.2488370](https://doi.org/10.1145/2487726.2488370).
- [47] J.-C. Huang, M. Monchiero, Y. Turner, and H.-H. S. Lee. Ally: OS-transparent packet inspection using sequestered cores. In *7th IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–11, 2011. doi: [10.1109/ANCS.2011.11](https://doi.org/10.1109/ANCS.2011.11).
- [48] Intel. Resources and response to side channel variants 1, 2, 3, Aug. 2018. <https://www.intel.com/content/www/us/en/architecture-and-technology/side-channel-variants-1-2-3.html>.
- [49] *Intel Virtualization Technology for Directed I/O Architecture Specification*. Intel, Apr. 2021. Order number D51397-013, rev. 3.3 <https://www.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html>.
- [50] *Software Guard Extensions Programming Reference*. Intel Corp., Oct. 2014. Ref. #329298-002 <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>.

- [51] *Intel SGX*. Intel Corp., June 2015. Ref. #332680-002 <https://www.intel.com/content/dam/develop/external/us/en/documents/332680-002-610985.pdf>.
- [52] *Control-flow Enforcement Technology Preview*. Intel Corp., June 2016. Ref. #334525-001 <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>.
- [53] *Intel Trust Domain CPU Architectural Extensions*. Intel Corp., Sept. 2020. Ref. #343754-001US <https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-cpu-architectural-specification.pdf>.
- [54] *Intel TDX Connect Architecture Specification*. Intel Corp., May 2021. <https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html>.
- [55] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 272–283, 2011. doi: [10.1145/2155620.2155652](https://doi.org/10.1145/2155620.2155652).
- [56] S. Johnson. Intel SGX and side-channels. Intel Developer Zone, Feb. 2018. <https://web.archive.org/web/20200228140427/https://software.intel.com/en-us/articles/intel-sgx-and-side-channels>.
- [57] V. Kanchanahalli. Power your Azure GPU workstations with flexible GPU partitioning. Azure Blog, Mar. 2020. <https://azure.microsoft.com/en-us/blog/power-your-azure-gpu-workstations-with-flexible-gpu-partitioning>.
- [58] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th IEEE International Symposium on Computer Architecture*, pages 350–361, June 2010. doi: [10.1145/1815961.1816010](https://doi.org/10.1145/1815961.1816010).
- [59] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st IEEE International Symposium on Computer Architecture*, pages 361–372, 2014. doi: [10.1109/ISCA.2014.6853210](https://doi.org/10.1109/ISCA.2014.6853210).
- [60] P. Kocher. Conference presentation of Kocher et al. [62], May 2019. <https://youtu.be/zOvBHxMjNls>.
- [61] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th International Cryptology Conference*, pages 388–397. Springer, Aug. 1999. doi: [10.1007/3-540-48405-1\\_25](https://doi.org/10.1007/3-540-48405-1_25).
- [62] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 1–19, 2019. doi: [10.1109/SP.2019.00002](https://doi.org/10.1109/SP.2019.00002).
- [63] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. Intel application note 321211–002, Jan. 2011. <https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf>.
- [64] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th ACM European Conference on Computer Systems*, pages 1–16, Apr. 2020. doi: [10.1145/3342195.3387532](https://doi.org/10.1145/3342195.3387532).
- [65] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium*, pages 557–574, 2017. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho>.
- [66] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. MyCloud: Supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th ACM Annual Computer Security Applications Conference*, pages 59–68, 2013. doi: [10.1145/2523649.2523680](https://doi.org/10.1145/2523649.2523680).
- [67] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected I/O operations in AMD’s secure encrypted virtualization. In *Proceedings of the 28th USENIX Security Symposium*, pages 1257–1272, Aug. 2019. ISBN 978-1-939133-06-9. <https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan>.
- [68] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, pages 1541–1541, 2022. doi: [10.1109/SP46214.2022.9833768](https://doi.org/10.1109/SP46214.2022.9833768).
- [69] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the Arm confidential compute architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, July 2022. <https://www.usenix.org/conference/osdi22/presentation/li>.

- [70] J. T. Lim and J. Nieh. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 557–574, 2020. ISBN 9781450371025. doi: [10.1145/3373376.3378467](https://doi.org/10.1145/3373376.3378467).
- [71] *The Devicetree Specification*. Linaro, 0.4-rc1 edition, Nov. 2021. <https://www.devicetree.org/specifications>.
- [72] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Aug. 2018. ISBN 978-1-939133-04-5. <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
- [73] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiawicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009. [https://www.usenix.org/legacy/events/hotpar09/tech/full\\_papers/liu/liu.pdf](https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/liu/liu.pdf).
- [74] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafir. DAMN: Overhead-free IOMMU protection for networking. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–315, Mar. 2018. doi: [10.1145/3173162.3173175](https://doi.org/10.1145/3173162.3173175).
- [75] R. J. Masti, C. Marforio, K. Kostianen, C. Soriente, and S. Capkun. Logical partitions on many-core platforms. In *Proceedings of the 31st ACM Annual Computer Security Applications Conference*, pages 451–460, 2015. doi: [10.1145/2818000.2818026](https://doi.org/10.1145/2818000.2818026).
- [76] *Hyper-V HyperClear Mitigation for L1 Terminal Fault*. Microsoft, Aug. 2018. <https://techcommunity.microsoft.com/t5/virtualization/hyper-v-hyperclear-mitigation-for-l1-terminal-fault/ba-p/382429>.
- [77] *Managing Hyper-V hypervisor scheduler types: The core scheduler*. Microsoft, Dec. 2021. <https://docs.microsoft.com/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types#the-core-scheduler>.
- [78] *About Azure DCasv5/ECasv5-series confidential virtual machines*. Microsoft Azure, Nov. 2021. <https://docs.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview>.
- [79] *Maintenance for virtual machines in Azure*. Microsoft Azure, Oct. 2021. <https://docs.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates>.
- [80] *B-series burstable virtual machine sizes*. Microsoft Azure, June 2022. <https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable>.
- [81] *Azure compute unit*. Microsoft Azure, Apr. 2022. <https://docs.microsoft.com/en-us/azure/virtual-machines/acu>.
- [82] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017. <https://eprint.iacr.org/2017/618>.
- [83] L. Nelson and X. Wang. Developing security monitors on RISC-V: Case studies on HiFive Unleashed. Technical Report UW-CSE-2019-11-01, University of Washington, Nov. 2019. URL <https://unsat.cs.washington.edu/papers/nelson-hifive-tr.pdf>.
- [84] K. T. Nguyen. Usage models for cache allocation technology in the Intel Xeon processor E5 v4 family, Feb. 2016. <https://www.intel.com/content/www/us/en/developer/articles/technical/cache-allocation-technology-usage-models.html>.
- [85] U. G. A. Office. Solarwinds cyberattack demands significant federal and private-sector response, Apr. 2021. <https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic>.
- [86] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida. Secure page fusion with VUsion. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 531–545, 2017. doi: [10.1145/3132747.3132781](https://doi.org/10.1145/3132747.3132781).
- [87] T. Palit, Y. Shen, and M. Ferdman. Demystifying cloud benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, Apr. 2016. doi: [10.1109/ISPASS.2016.7482080](https://doi.org/10.1109/ISPASS.2016.7482080).
- [88] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi. Fast local page-tables for virtualized NUMA servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021. doi: [10.1145/3445814.3446709](https://doi.org/10.1145/3445814.3446709).
- [89] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*, pages 565–581, 2016. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl>.



- [90] I. Puddu, M. Schneider, M. Haller, and S. Čapkun. Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In *Proceedings of the 30th USENIX Security Symposium*, pages 663–680, 2021. <https://www.usenix.org/conference/usenixsecurity21/presentation/puddu>.
- [91] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, May 2021. doi: [10.1109/SP40001.2021.00020](https://doi.org/10.1109/SP40001.2021.00020).
- [92] Research and Markets. Bare metal cloud market by service type, organization size, vertical, and region – global forecast to 2026, Apr. 2021. Report number 5316699 <https://www.researchandmarkets.com/reports/5316699/bare-metal-cloud-market-by-service-type-compute>.
- [93] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, June 2019. Ver. 20190608-Priv-MSU-Ratified <https://riscv.org/risc-v-isa>.
- [94] *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. RISC-V International, Jan. 2021. <https://github.com/riscv/opensbi>.
- [95] *OpenSBI Domain Support*. RISC-V OpenSBI, Nov. 2020. [https://github.com/riscv/opensbi/blob/c0d2baa/docs/domain\\_support.md](https://github.com/riscv/opensbi/blob/c0d2baa/docs/domain_support.md).
- [96] J. R. Sanchez Vicarte, P. Shome, N. Nayak, C. Tripel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher. Opening Pandora’s Box: A systematic study of new ways microarchitecture can leak private data. In *Proceedings of the 48th IEEE International Symposium on Computer Architecture*, pages 347–360, 2021. doi: [10.1109/ISCA52012.2021.00035](https://doi.org/10.1109/ISCA52012.2021.00035).
- [97] J. R. Sanchez Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022. doi: [10.1109/SP46214.2022.00089](https://doi.org/10.1109/SP46214.2022.00089).
- [98] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security*, pages 753–768, 2019. doi: [10.1145/3319535.3354252](https://doi.org/10.1145/3319535.3354252).
- [99] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 335–350, 2007. doi: [10.1145/1294261.1294294](https://doi.org/10.1145/1294261.1294294).
- [100] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A thin hypervisor for enforcing I/O device security. In *Proceedings of the 5th ACM International Conference on Virtual Execution Environments*, pages 121–130, 2009. doi: [10.1145/1508293.1508311](https://doi.org/10.1145/1508293.1508311).
- [101] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, pages 317–328, 2016. doi: [10.1145/2897845.2897885](https://doi.org/10.1145/2897845.2897885).
- [102] *SiFive FU540-C000 Manual*. SiFive, 1.0 edition, Apr. 2018. <https://static.dev.sifive.com/FU540-C000-v1.0.pdf>.
- [103] *SiFive WorldGuard White Paper, v1.2*. SiFive, Dec. 2020. [https://sifive.cdn.prismic.io/sifive/aa27fffb-cf24-4077-8103-682f26141b69\\_WorldGuard\\_White\\_Paper\\_v1.2.pdf](https://sifive.cdn.prismic.io/sifive/aa27fffb-cf24-4077-8103-682f26141b69_WorldGuard_White_Paper_v1.2.pdf).
- [104] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th IEEE International Symposium on Computer Architecture*, pages 318–331, 2019. doi: [10.1109/MM.2020.2986204](https://doi.org/10.1109/MM.2020.2986204).
- [105] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th ACM European Conference on Computer Systems*, pages 209–222, 2010. doi: [10.1145/1755913.1755935](https://doi.org/10.1145/1755913.1755935).
- [106] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 437–450, 2012. doi: [10.1145/2150976.2151022](https://doi.org/10.1145/2150976.2151022).
- [107] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 401–412, 2011. doi: [10.1145/2046707.2046754](https://doi.org/10.1145/2046707.2046754).
- [108] B. Teabe, P. Yuhala, A. Tchana, F. Hermenier, D. Hagimont, and G. Muller. (No)Compromis: Paging virtualization is not a fatality. In *Proceedings of the 17th ACM International Conference on Virtual Execution Environments*, pages 43–56, 2021. doi: [10.1145/3453933.3454013](https://doi.org/10.1145/3453933.3454013).



- [109] *Advanced Configuration and Power Interface (ACPI) Specification*. UEFI Forum, 6.4 edition, Jan. 2021. <https://uefi.org/specifications>.
- [110] J. Van Bulck, F. Piessens, and R. Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017. ISBN 978-1-4503-5097-6. doi: [10.1145/3152701.3152706](https://doi.org/10.1145/3152701.3152706).
- [111] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, pages 1041–1056, 2017. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck>.
- [112] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, 2018. ISBN 978-1-939133-04-5. <https://www.usenix.org/conference/usenixsecurity18/presentation/bulck>.
- [113] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, pages 178–195, 2018. doi: [10.1145/3243734.3243822](https://doi.org/10.1145/3243734.3243822).
- [114] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 88–105, May 2019. doi: [10.1109/SP.2019.00087](https://doi.org/10.1109/SP.2019.00087).
- [115] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. SGAXe: How SGX fails in practice, 2020. <https://sgaxe.com/files/SGAXe.pdf>.
- [116] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, pages 339–354, 2021. doi: [10.1109/SP40001.2021.00064](https://doi.org/10.1109/SP40001.2021.00064).
- [117] C. Waldspurger and M. Rosenblum. I/O virtualization. *Communications of the ACM*, 55(1):66–73, Jan. 2012. doi: [10.1145/2063176.2063194](https://doi.org/10.1145/2063176.2063194).
- [118] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002. <https://www.usenix.org/legacy/events/osdi02/tech/waldspurger/waldspurger.pdf>.
- [119] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. Using burstable instances in the public cloud: Why, when and how? *Proceedings of ACM on Measurement and Analysis of Computing Systems*, 1(1), June 2017. doi: [10.1145/3084448](https://doi.org/10.1145/3084448).
- [120] X. Wang, Y. Shi, Y. Dai, Y. Qi, J. Ren, and Y. Xuan. TrustOSV: Building trustworthy executing environment with commodity hardware for a safe cloud. *Journal of Computers*, 9(10):2303–2314, Oct. 2014. ISSN 1796-203X. <http://www.jcomputers.us/vol9/jcp0910-07.pdf>.
- [121] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 127–140, 2012. ISBN 9781450312233. doi: [10.1145/2168836.2168850](https://doi.org/10.1145/2168836.2168850).
- [122] D. Wentzlaff, C. J. Jackson, P. Griffin, and A. Agarwal. Configurable fine-grain protection for multicore processor virtualization. In *Proceedings of the 39th IEEE International Symposium on Computer Architecture*, pages 464–475, 2012. doi: [10.1109/ISCA.2012.6237040](https://doi.org/10.1109/ISCA.2012.6237040).
- [123] R. West, Y. Li, E. Missimer, and M. Danish. A virtualized separation kernel for mixed-criticality systems. *ACM Transactions on Computer Systems*, 34(3), June 2016. doi: [10.1145/2935748](https://doi.org/10.1145/2935748).
- [124] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th ACM International Conference on Virtual Execution Environments*, pages 31–40, 2009. ISBN 9781605583754. doi: [10.1145/1508293.1508299](https://doi.org/10.1145/1508293.1508299).
- [125] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprived execution. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, Feb. 2013. <https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/taming-hosted-hypervisors-mostly-deprived-execution>.
- [126] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture*, pages 246–257, 2013. doi: [10.1109/HPCA.2013.6522323](https://doi.org/10.1109/HPCA.2013.6522323).

- [127] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom. Dominance as a new trusted computing primitive for the internet of things. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 1415–1430, 2019. doi: [10.1109/SP.2019.00084](https://doi.org/10.1109/SP.2019.00084).
- [128] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side-channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 640–656, May 2015. doi: [10.1109/SP.2015.45](https://doi.org/10.1109/SP.2015.45).
- [129] X. Zhang, X. Zheng, Z. Wang, Q. Li, J. Fu, Y. Zhang, and Y. Shen. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–105, 2019. doi: [10.1145/3297858.3304034](https://doi.org/10.1145/3297858.3304034).





# ExoFlow: A Universal Workflow System for Exactly-Once DAGs

Siyuan Zhuang  
*UC Berkeley*

Stephanie Wang  
*UC Berkeley, Anyscale*

Eric Liang  
*Anyscale*

Yi Cheng  
*Anyscale*

Ion Stoica  
*UC Berkeley*

## Abstract

Given the fundamental tradeoff between run-time and recovery performance, current distributed systems often build application-specific recovery strategies to minimize overheads. However, it is increasingly common for different applications to be composed into heterogeneous pipelines. Implementing multiple interoperable recovery techniques in the same system is rare and difficult. Thus, today’s users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) the challenging task of stitching together multiple systems that can offer application-specific tradeoffs.

We present ExoFlow, a *universal workflow* system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. The key insight behind our solution is to *decouple execution from recovery* and provide exactly-once semantics as a separate layer from execution. For generality, workflow tasks can return *references* that capture arbitrary inter-task communication. To enable the workflow system and therefore the end user to take control of recovery, we design task annotations that specify execution semantics such as nondeterminism. ExoFlow generalizes recovery for existing workflow applications ranging from ETL pipelines to stateful serverless workflows, while enabling further optimizations in task communication and recovery.

## 1 Introduction

A key requirement for distributed applications is *fault tolerance*, i.e. the appearance of execution without failures even when failures occur. In general, there is a tradeoff between recovery and run-time overhead. For example, logging generally adds higher execution overhead but reduces recovery time by allowing the system to only re-execute computations that failed [23]. Meanwhile, checkpointing reduces execution overhead but can impose higher recovery overhead as the system must roll back additional computation after a failure.

Current distributed systems often choose different tradeoff points between recovery and performance based on the application. For example, Apache Spark uses lineage-based logging for batch processing [48], and Apache Flink uses checkpointing for stream processing [19].

However, it is becoming increasingly common for different applications to be composed into heterogeneous pipelines. For example, a machine learning pipeline might use batch ingest to build a training dataset, then stream the data to a

batch distributed training job to reduce latency and memory overhead. If we use a single recovery strategy for the entire pipeline, performance and recovery may be suboptimal because different recovery strategies are suited to different applications. Thus, to optimize end-to-end performance and recovery, we need to *compose different recovery strategies*.

Implementing multiple, interoperable recovery techniques within the same system, let alone a single one, is challenging. For example, Spark introduced “continuous processing” to reduce performance overheads for stream processing applications, but this mode does not yet provide exactly-once semantics during failures [10]. On the other hand, Flink has added a batch processing mode, but this required building an entirely separate recovery system from the streaming path [20].

Overall, these challenges have led to patchy support for applications that have diverse requirements in the recovery-performance tradeoff space. Users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) stitching together multiple systems that offer different application-specific tradeoffs. The latter, however, is challenging and requires coordinating the flow of data, control, and recovery across disparate systems. This is true even in a single system, if using disparate execution modes such as batch vs. streaming.

In this paper, we propose a *universal workflow* system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. A *workflow* is a directed acyclic graph (DAG) of *tasks*, where each task encapsulates a function call and edges between tasks represent data dependencies. Workflows are used to orchestrate execution across systems and thus prioritize generality. The DAG API is popular because it allows arbitrary application code in each task, from submitting a Spark job to invoking a microservice.

In contrast to other workflow systems, however, we *decouple the unit of execution from the unit of recovery*. In particular, ExoFlow guarantees fault tolerance by durably logging the workflow DAG and coordinating task checkpoint and recovery, while execution of the DAG is handled by a generic “backend”. This has three key benefits. First, it enables heterogeneous application pipelines that need multiple recovery strategies for performance. Second, it augments existing distributed execution frameworks that provide only at-most-once or at-least-once semantics with strong exactly-once semantics. Third, it disaggregates the execution backend from recovery, allowing independent deployment and scaling.



Previous workflow systems provide exactly-once semantics but with significant limitations. For generality, workflow systems such as Apache Airflow [3] assume that each task is nondeterministic and may have side effects on external systems that in general cannot be rolled back. Thus, each task must synchronously checkpoint its outputs *before* they can be made visible to any downstream tasks. Otherwise, the system may have to re-execute the task in case of a failure. If the re-execution produces a different result, this can cause an inconsistent view among downstream tasks and external systems.

Thus, by assuming the worst, the workflow system has only one option of ensuring fault tolerance: no task can start before its upstream tasks have finished checkpointing all of their outputs. This limits the workflow system’s ability to incorporate key optimizations often employed by application-specific frameworks that exploit the application’s semantics. For example, large datasets passed between tasks can often be deterministically regenerated, making checkpointing unnecessary. In addition, while some tasks may indeed have external effects, e.g., starting a transaction on an external database, some effects can also be rolled back, e.g., by aborting the transaction.

Our goal is to hand control over recovery to ExoFlow and ultimately the end user. Thus, we use two key interfaces to enable awareness of application semantics. First, we extend the typical workflow DAG API with pluggable *first-class references* to enable more flexible workflow-internal communication. A workflow task can return references to its outputs, which the workflow system then passes to downstream tasks. In contrast, current workflow systems require the application to pass data by explicitly copying and checkpointing, which can be expensive for large data, or implicitly through external storage, which makes it difficult to guarantee exactly-once semantics. By using references to capture arbitrary data movement between workflow tasks, ExoFlow leverages third-party systems’ existing communication and recovery mechanisms while retaining control over workflow-level recovery.

Second, we introduce user annotations that specify relevant task semantics, i.e. whether to checkpoint a task, whether the outputs are deterministic, and whether the task has externally visible outputs. Before execution, ExoFlow checks the safety of the user’s specification. During execution, ExoFlow synchronizes task execution and checkpointing. During recovery, ExoFlow coordinates *rollback*, e.g., deletion of outputs from a previous execution, and task replay. For example, before executing a task with an externally visible output, ExoFlow will first synchronize upstream checkpoints to *commit* any nondeterministic outputs, i.e. ensure they will never be rolled back. This allows the user to flexibly and safely optimize the recovery technique.

ExoFlow is built on Ray [37] and consists of a per-workflow centralized controller, a pluggable checkpoint storage, and a pluggable execution backend. Centralizing controller logic makes it simple to guarantee recovery correctness. Meanwhile, checkpointing and execution are fully disaggregated,

allowing these to be scaled independently of the controller.

We demonstrate the benefits of ExoFlow with two execution backends, the Ray framework and AWS Lambdas, both distributed frameworks that provide at-most-once or at-least-once tasks. We show that references can enable  $\sim 5\times$  speedup for Spark data processing workflows compared to Apache Airflow, while task annotations enable 51% lower latency for transactional serverless workflows compared to Beldi [49]. These optimizations are possible because correctness is ultimately guaranteed by ExoFlow. These results also demonstrate ExoFlow’s *universality*, as the system is not specific to data processing or serverless environments. In summary, our contributions are:

1. Decoupling execution from recovery to enable a flexible tradeoff between performance and fault tolerance.
2. Designing a universal workflow system that guarantees exactly-once DAG execution.
3. Demonstrating benefits for a diverse set of applications, including an ML pipeline, serverless transactions, and graph processing that mixes stream and batch execution.

## 2 Motivation

### 2.1 Overview of recovery strategies

We use *exactly-once semantics* as our correctness condition. This condition often implies application-specific correctness properties, such as global consistency in message-passing systems [23] or linearizability in storage systems [29].

More precisely, exactly-once semantics require all outputs to appear consistent with a physical execution where all inputs were processed without failures. In a workflow setting, the inputs are the DAG and the root task arguments. Outputs are values produced by a task that are viewed by others.

Output visibility can be *internal* or *external*. For example, values passed between tasks in Figure 1a are internal because they are viewed only by other tasks. Meanwhile, `(key, val)` is external because it is sent to a key-value store. Once outputs are made external, the workflow system no longer has control over how they will be used, e.g., via reads from external key-value store clients. Outputs can also be either *deterministically* or *nondeterministically* generated.

Output visibility and determinism are important because together they determine the recovery procedures that will guarantee exactly-once semantics (Figure 1b). For example, consider the cases if A is nondeterministic and we do not checkpoint `a_out` in Figure 1a. Suppose C views an initial value `a_out1` and produces `c_out1`, but we lose `a_out1` due to a failure. If we re-execute A to produce `a_out2` and pass this to B, the outputs of B and C will not be consistent with a failure-free execution. To handle this case, we also need to “rollback” `c_out1` and re-execute C on `a_out2`.

We encounter additional problems in the opposite case where B finishes and we then lose `a_out1`. B has already made `(key, val)` external and these values may depend on `a_out1`.

If we execute  $C$  on  $a\_out_2$ ,  $c\_out$  will be inconsistent with  $(key, val)$ . Thus, the only way to guarantee correctness in this case is to either: (1) “commit”  $a\_out_1$  before executing  $B$ , e.g., by checkpointing it, or (2) gain application semantics about how to roll back visibility of  $(key, val)$ .

Meanwhile, deterministic outputs are safe to view as long as the task can be replayed on its original inputs and recomputed outputs can be deduplicated. The external output in Figure 1a can for example be deduplicated by attaching a deterministic `req_id`.

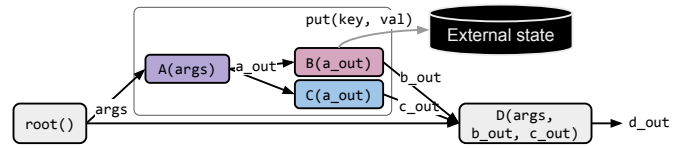
**Solution space.** Handling nondeterministic outputs is generally done in two ways: (1) global checkpointing and rollback on failure, or (2) logging and deterministic replay on failure [23]. Both “commit” a prefix of a failure-free execution by saving the outputs of a task frontier, allowing recovery to resume execution from a consistent set of intermediate outputs. Global checkpointing advances this frontier several tasks at a time and upon failure, rolls back to the last frontier to undo partially visible nondeterministic outputs. For outputs that cannot be rolled back, however, upstream nondeterministic outputs must first be committed by taking a global checkpoint. Logging-based methods advance the frontier one task at a time by committing each nondeterministic output before making it visible, thus avoiding additional rollback on failure.

Note that rollback and durability options vary based on output visibility. External outputs may be impossible to roll back, e.g., a transaction commit cannot be undone, or make durable, as third-party system context is not always serializable.

Current workflow systems guarantee exactly-once semantics by: (1) durably checkpointing each internal output before making it visible, and (2) requiring the developer to make external outputs idempotent and durable. This one-size-fits-all approach does not leverage application-specific recovery methods (Figure 1b). Furthermore, existing workflow systems have fundamental limits on internal outputs, usually because they must be sent between tasks through the workflow controller. Apache Airflow uses a database to coordinate tasks, which imposes a maximum output size on the order of MBs [3], and direct task communication in FaaS is limited [24]. Together, these force developers to use external outputs for much of their task communication [24, 42].

Our goal is to support different recovery methods in a single workflow system and even within a single application. The key insight behind ExoFlow is that knowing the DAG structure makes it simple to identify a consistent execution frontier, allowing the recovery methods before and after the frontier to be decoupled. For example,  $a\_out$  is internal to the outlined sub-DAG in Figure 1a and thus its recovery method can be chosen flexibly as long as the inputs ( $args$ ) and outputs ( $b\_out, c\_out, key, val$ ) are consistent.

Thus, our solution consists of two parts. First, *references* enable ExoFlow to capture a broader range of inter-task communication as internal outputs, without being involved in the physical communication. This encourages recovery



(a) Workflow DAG

|                         | Internal                                                 | External                                                                              |
|-------------------------|----------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>Nondeterministic</b> | Commit output OR on failure, rollback visibility         | Commit output <i>before</i> visibility OR if possible, rollback visibility on failure |
| <b>Deterministic</b>    | Replay failed task(s) on previous inputs, dedupe outputs | Also dedupe external outputs                                                          |

(b) Recovery strategies for workflow DAGs

Figure 1: (a) An example workflow with internal outputs (e.g.,  $a\_out$ ) and external outputs (e.g., `put(key, val)`). (b) The most efficient recovery strategy depends on output visibility and nondeterminism.

flexibility within a sub-DAG and recovery independence across sub-DAGs. References enable efficient passing of task outputs of any size and location as well as outputs that may not be serializable.

Second, we support *annotations* to specify task semantics (checkpointing, nondeterminism, output visibility). These allow the system to determine recovery correctness before execution. The system “commits” the application to this specification by durably logging the DAG before execution, then coordinates and synchronizes task checkpoints during execution.

## 2.2 Applications

We use three representative applications to show the value of: (1) making workflow-internal outputs more flexible, and (2) exposing application semantics to the workflow controller:

1. Extract-transform-load (ETL) pipelines: Using references to pass large data as internal outputs.
2. Machine learning (ML) pipelines: Using references to pass large data and leveraging application semantics.
3. Serverless workflows: Leveraging application semantics to reduce recovery overheads, in a way that is agnostic to external systems.

**ETL pipelines.** Workflow systems such as Apache Airflow are commonly used to orchestrate extract-transform-load (ETL) pipelines composed of data processing jobs. Figure 2a shows an example in which a Spark job  $A$  performs batch data cleaning and writes the data to an external database, e.g., Delta Lake [11]. Jobs  $B$  and  $C$  then load the data for querying.

Current practice for exactly-once workflow execution requires all of  $A$ ’s outputs to be made durable *before* executing  $B$  and  $C$ . Synchronous checkpointing adds high overhead for large and distributed data. In addition,  $B$  and  $C$  must each reload the data, imposing an unnecessary memory copy. This is of course unnecessary if  $A$  is deterministic. Execution systems such as Spark leverage this property to natively support distributed in-memory caching. Ideally,  $A$  should pass its output as a cached RDD [48] to  $B$  and  $C$  (Figure 2b), avoiding the round trip to external storage, allowing  $B$  and  $C$  to share

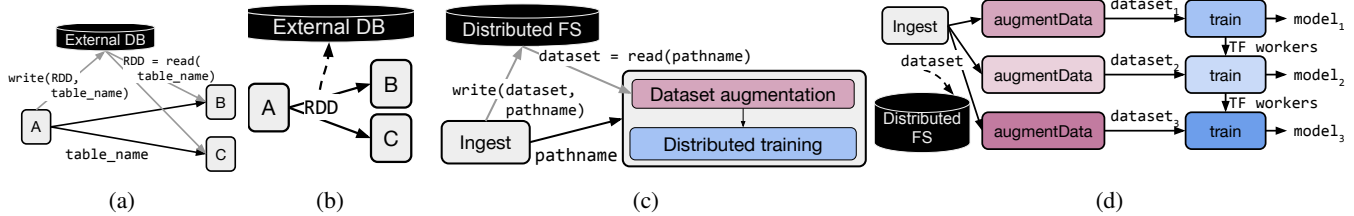


Figure 2: (a) ETL workflow today, using external outputs for communication. (b) The same ETL workflow with internal outputs only. (c) ML training workflow today, with external outputs and manual orchestration within a task. (d) The same ML workflow with internal outputs only, and orchestration is handled by the workflow system. Third-party framework state (TF workers) can be passed between workflow tasks.

physical memory, and enabling asynchronous checkpointing.

Building such optimizations into a workflow system would enable orchestration of arbitrary DAGs and third-party frameworks. However, even with awareness of task determinism, current workflow systems cannot execute Figure 2b due to limitations in workflow-internal data passing.

**ML pipelines.** Machine learning (ML) pipelines are similar to ETL pipelines, but with an ML application as the end consumer. This requires composition of traditional ETL systems with distributed ML frameworks for training and inference. Figure 2c shows a typical ML training workflow, in which training data is extracted and transformed in the Ingest task, then consumed by a distributed training job. Loading data into the training job may itself require complex and possibly distributed data processing, with computations such as random transforms to augment datasets [40]. Furthermore, datasets are often large enough that preprocessing must be pipelined with training to maximize GPU utilization.

Current workflow systems cannot effectively orchestrate within the training task, as training data and worker state must be passed through distributed memory. Expanding workflow-internal outputs would enable workflows such as Figure 2d. To reduce the overhead of recovery, however, the workflow system also requires application semantics, such as whether dataset augmentation is deterministic. Also, the model output can be consumed in a variety of ways, from local one-off testing during development to deployment on an ML serving system during production. All of these factors affect the optimal correct recovery strategy.

**Serverless workflows.** In the functions-as-a-service (FaaS) model, the user breaks their application into small functions that can be transparently executed and scaled without explicit resource provisioning. Serverless functions have a limited lifetime, all local state is transient, and failure handling is usually limited to function retries. This makes it challenging to build fault-tolerant nontrivial applications directly on FaaS [28].

Recently, *serverless workflow* systems [16, 46, 49] have gained popularity as a solution, especially for stateful applications. A common strategy for guaranteeing exactly-once execution is to provide fault-tolerant APIs to capture external outputs. For example, Figure 3 shows an example of a trip reservation workflow [25] that places the order if and only if both the hotel and flight were successfully reserved. Systems such as Aft [46], Beldi [49], and Boki [32] guarantee exactly-

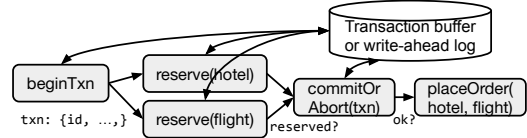


Figure 3: Serverless workflow systems [32, 46, 49] guarantee exactly-once semantics by interposing on all communication to external storage, e.g., through a transaction buffer, and explicitly managing visibility of these external effects.

once semantics by providing a transactional key-value store to manage external output visibility.

However, each system offers different isolation levels that require different recovery strategies. Aft buffers uncommitted writes, which are safe to rollback, while Beldi and Boki use write-ahead logging. Thus, each system implements their own recovery procedures, e.g., durability and task re-execution.

ExoFlow factors out workflow recovery to enable flexibility and optimizations. Instead of providing opinionated APIs for external outputs, we treat external systems such as the transaction buffer in Figure 3 as a black box. ExoFlow does not interpose on the communication to this external system and instead requires that the application can specify task semantics such as whether the external effect can be rolled back. These semantics can be specified by a particular transaction system, i.e. Aft or Beldi.

### 3 API

#### 3.1 Overview and requirements

ExoFlow is a general workflow layer that abstracts a workflow *backend*, i.e. a distributed framework providing at-least-once and/or at-most-once remote function invocation. We overview the application-facing API (Table 1) and requirements. The application must be able to: (1) differentiate deterministic tasks, and (2) for tasks with external outputs, ensure that the task is idempotent or specify an idempotent rollback function.

**DAG interface.** The application invokes workflow tasks and specifies arguments using `f.bind` (Table 1). The caller receives a `WorkflowDAG` that represents the task’s output and that can be passed to other tasks as dependencies. Workflow execution is *lazy*: to evaluate a `WorkflowDAG`, the developer must run it. This is to simplify recovery, as the workflow system can check DAG-level properties before executing it.

The workflow backend should implement an RPC-like inter-



| Workflow API                                                         | Semantics                                                                                                                                                                                                                                                                                  |
|----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>f.options(Opts).bind(Value   WorkflowDAG) → WorkflowDAG</code> | Create a workflow task <code>f</code> . Creates and returns a <code>WorkflowDAG</code> , whose value is lazily evaluated. The caller may pass the <code>WorkflowDAG</code> to another task. The return value of <code>f</code> can be a <code>WorkflowDAG</code> , i.e. a nested workflow. |
| <code>run(WorkflowDAG w, str name) → Value</code>                    | Run the workflow <code>w</code> and return the result. Optionally take a string identifier for this workflow.                                                                                                                                                                              |
| <code>run_async(WorkflowDAG w, str name) → Fut</code>                | Run the workflow <code>w</code> asynchronously and return a future that can be used to retrieve the result.                                                                                                                                                                                |
| <code>Ref.get() → Value</code>                                       | Used by the application to dereference to a value. Ref construction is backend-specific.                                                                                                                                                                                                   |
| <code>bool Opts.checkpoint=True</code>                               | True if the task's output should be saved.                                                                                                                                                                                                                                                 |
| <code>bool Opts.deterministic=False</code>                           | True if outputs are deterministically generated.                                                                                                                                                                                                                                           |
| <code>bool Opts.can_rollback=False</code>                            | True if task has no external outputs, or if they can be rolled back. If False, the task must be idempotent.                                                                                                                                                                                |
| <code>Fn Opts.rollback=null</code>                                   | If external outputs can be rolled back, a function to do so. The function must be idempotent, and any <code>WorkflowDAG</code> arguments must be a subset of the original workflow task <code>f</code> 's arguments.                                                                       |
| <code>Ref._id() → ID</code>                                          | Used by the workflow system to compare equality.                                                                                                                                                                                                                                           |
| <code>Ref._checkpoint() → Fut[Value]</code>                          | Used by the workflow system to coordinate checkpointing. The <code>Value</code> is the checkpoint data or metadata.                                                                                                                                                                        |
| <code>Ref._restore(Value)</code>                                     | Used by the workflow system to reload from a saved checkpoint.                                                                                                                                                                                                                             |

Table 1: Workflow API. Top: API calls exposed to the application. Middle: Task annotations specified by application or third-party library. Bottom: ExoFlow-internal Ref API, pluggable by execution backend.

face. Within a task, the application can invoke arbitrary local or distributed execution. For greater generality, we also adopt the *dynamic* task model [39]: tasks can dynamically invoke exactly-once nested workflows by returning a `WorkflowDAG`.

**Task annotations.** The application specifies semantics relevant to recovery at task invocation time (Table 1). The workflow system uses these to ensure correctness of: (1) coordination of distributed workflow checkpoints during execution, and (2) output rollback and task re-execution upon failure.

First, the application specifies whether to skip checkpointing a task's output. Note that the workflow system guarantees correctness, so this can be considered an optimization hint, e.g., to avoid recomputation for long tasks,

Next, the application can specify whether a task's outputs (both internal and external) are deterministic. This allows the workflow system to minimize rollback during recovery.

Finally, the application specifies whether a task can be rolled back and if yes, how to do so. Tasks with no external outputs, such as the data processing tasks in Figure 2, should set `can_rollback=True`. Tasks that have external outputs that cannot be rolled back should set `can_rollback=False` and ensure idempotence, as recovery may require re-execution.

Non-idempotent tasks with external outputs that can be rolled back should set `can_rollback=True` and the `rollback` callback. On failure, ExoFlow executes these rollback "tasks" in reverse dependency order before resuming execution. The rollback task can take any arguments available to the original workflow task, but the application must additionally guarantee that the rollback task is idempotent. For example, to implement the transaction in Figure 3, rollback for the `beginTxn` and `reserve` tasks could simply abort.

On run, ExoFlow checks the `WorkflowDAG` for specification errors and throws an exception if any are found. In particular, correctness requires the application to set checkpoints between each nondeterministic task and each downstream task with external output. Section 3.3 makes this precise.

**Internal outputs.** Direct task outputs are subject to limits of the execution backend. For greater flexibility, ExoFlow

allows outputs to include `Refs` created by the task. `Refs` are (optionally) pluggable by the execution backend. They are intended to capture volatile outputs that would be expensive or complex to natively support in ExoFlow, e.g., large distributed data or third-party framework context. For an AWS Lambdas backend, for example, values can be stored in an external (volatile) key-value store and the key can be passed in a `Ref`. Other tasks can dynamically get the value, which can throw an error if the value is irretrievable due to failure.

`Refs` are uniquely identifiable objects typically containing backend-specific metadata. A task can only return `Refs` that it created or that were passed to it by an upstream task. Then, upon failure, ExoFlow can either restore the `Ref` from a checkpoint, or trace the DAG back to the creating task. On re-execution, the task need not return the same `Refs` as its original execution. For example, with the annotation `deterministic=True`, it is only necessary that the *value* of a returned `Ref` is deterministic; the `Ref` itself may have a nondeterministic ID. This is safe because ExoFlow simply cancels tasks using the previous `Refs` and re-executes with the new `Refs`.

By default, `Ref` values are *immutable*. This improves recovery efficiency, as it simplifies checkpointing and minimizes task rollback. To capture task outputs that are expensive or impossible to materialize, we also support *stateful* references, i.e. *actors* [30]. An `ActorRef` extends `Refs` with application-defined methods that execute on the actor's state (Listing 1). However, mutable state is more complex to recover efficiently and correctly. Thus, compared to `Refs`, we limit how `ActorRefs` can be passed between workflow tasks (Section 3.4).

## 3.2 Model

We present a formal model of workflows to more precisely capture the API and assumptions. A *workflow*  $G = (V, E)$  is a directed acyclic graph with vertices  $V$  and edges  $E$ . Each vertex  $v_i$  has an associated function  $F_i$ , a function  $\mathcal{N}_i$  representing a (potential) source of nondeterminism, a nullable rollback function  $\mathcal{R}_i$ , and the annotations described in Table 1.

A workflow execution produces one internal and one



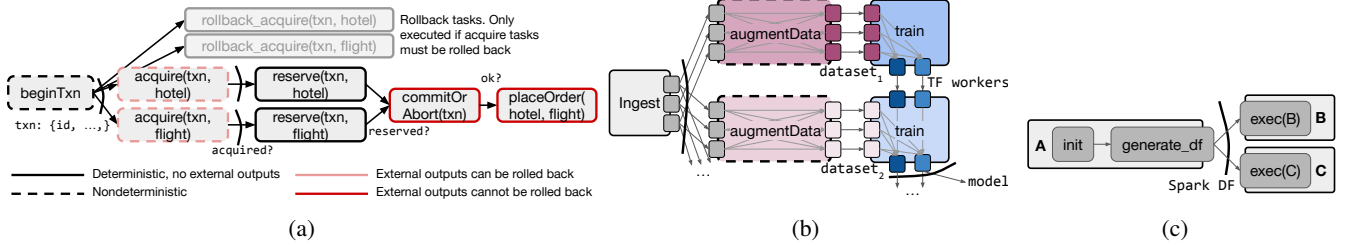


Figure 4: (a) Task annotations. Edge cuts represent `checkpoint=True`. (b) Passing references (small boxes) in an ML workflow. Blue Refs are actors that wrap TensorFlow worker state. (c) Passing an ActorRef in an ETL workflow. B and C call read-only methods on the Spark context actor.

external output per vertex, both nullable. For brevity, we do not consider tasks with multiple outputs. We denote an execution’s outputs by  $(O_{Int}, O_{Ext})$ , both mappings from vertex to a single output  $o_{Int}$  or  $o_{Ext}$ .  $F_i$  outputs  $o_{Ext}$  by adding it to a global set  $\mathcal{W}$ , which can be read by other tasks and external processes. Each  $F_i$  takes as inputs:

- $I_{Int}$ : Internal outputs of vertices with an edge to  $v_i$ .
- $w_i$ : A read of  $\mathcal{W}$ , i.e. the external outputs so far.
- $n_i$ : A nondeterministic value returned by  $\mathcal{N}_i$ .

In other words, an edge  $(v_i, v_j)$  indicates that  $v_i$ ’s internal output is passed to task  $v_j$ . Internal outputs passed between vertices are analogous to messages passed between processes in a message-passing model [23], except that the application must declare the “messages” (dependencies) before execution.

$\mathcal{N}_i$  captures nondeterministic inputs. For example, if  $F_i$  depends on the current time, then  $\mathcal{N}_i$  returns the current time. We assume that if  $\mathcal{N}_i$  reads some external state, the external state will not be rolled back (unless  $F_i$  is also rolled back via  $\mathcal{R}_i$ ).

The correctness condition relates outputs to a failure-free execution.  $W$  denotes all possible sequences of reads of  $\mathcal{W}$  by an external process.

**Definition 1** (Consistency).  $O_{Int}, O_{Ext}$  are consistent with a workflow  $G = (V, E)$  if  $W$  is monotonic and  $\forall w \in W$ :

$$o_{Ext} \in w \Rightarrow \exists i, O_{Ext}[i] = o_{Ext} \bigwedge (O_{Int}[i], o_{Ext}) = F_i([O_{Int}[j] \mid (v_j, v_i) \in E], \{O_{Ext}[j] \mid v_j <_G v_i\}, \mathcal{N}_i()) \bigwedge \{O_{Ext}[j] \mid v_j <_G v_i\} \subseteq w$$

More simply, from an external process’s perspective, if it sees an external output, then: (1) the same output was seen in all previous reads, (2) it must correspond to one invocation of some  $F_i$ , and (3) it also sees the external outputs of all predecessors of  $v_i$ . This is analogous to *global consistency* in message-passing [23], i.e. that every visible output has a corresponding task that created it. The goal is to provide a consistent execution under a crash failure model.

The application assumptions are as follows. For each  $v_i$ :

1. If  $v_j$  is concurrent with  $v_i$  ( $v_i \not\prec_G v_j$  and  $v_j \not\prec_G v_i$ ), then  $F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w_i \setminus \{O_{Ext}[j]\}, n_i)$ .
2. If `deterministic=True`, then  $\{O_{Ext}[j] \mid v_j <_G v_i\} \subseteq w_i, w'_i \Rightarrow F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w'_i, n'_i)$ .
3. If the  $o_{Ext}$  returned by  $F_i$  is not null, then either `can_rollback=False` or  $\mathcal{R}_i$  is not null.

- (a) If `can_rollback=False`, then  $F_i$  is idempotent. That is, if  $(o_{Int}, o_{Ext}) = F_i(I_{Int}, w, n_i)$  and  $(o'_{Int}, o'_{Ext}) = F_i(I_{Int}, w', n'_i)$ , then  $o_{Ext} = o'_{Ext}$ .
- (b) If  $\mathcal{R}_i$  is provided, then it is a deterministic and idempotent function of the task’s internal inputs. If  $(o_{Int}, o_{Ext}) = F_i(I_{Int}, w, n_i)$ , then  $\mathcal{R}_i(I_{Int})$  removes  $o_{Ext}$  from all past reads of  $\mathcal{W}$ .

(1) means that we do not consider cases in which a task  $v_i$  depends on a task  $v_j$ ’s external output, where  $v_i, v_j$  cannot be ordered in  $G$ . To ensure consistency,  $v_j$ ’s external output should be considered part of  $v_i$ ’s nondeterministic input, and  $v_j$  must set `can_rollback=False`. Regarding (3b), note that the meaning of removing  $o_{Ext}$  from past reads is application-dependent. For example, suppose  $F_i$  executes a transaction and  $\mathcal{R}_i$  aborts the transaction; if uncommitted reads are allowed, then  $\mathcal{R}_i$  does not need to roll back the reader.

**Nested tasks and references.** While not explicitly captured in the above model, nested tasks can be thought of as tasks that expand into a sub-workflow. Refs and ActorRefs are native data types that can be returned in a function’s internal output. Because actors are mutable, ActorRefs are versioned: if a caller writes to an actor by calling a method on its ActorRef, the caller’s resulting ActorRef is of a different version. This becomes relevant in Section 3.4, which discusses the rules that the application must follow to ensure exactly-once semantics when ActorRefs are passed between workflow tasks.

### 3.3 Guaranteeing exactly-once execution

Task annotations simplify the decision of when to commit task outputs. To illustrate this, we use Figure 4a, a modified version of the workflow described in Figure 3. We show the annotations for a workflow using an external two-phase locking (2PL) transaction system. `beginTxn` generates a transaction context with a random `txn_id`. The `acquire` tasks each attempt to acquire a lock on an external table row. If this is successful, we attempt to reserve the flight and hotel if available, then finally commit the transaction and place the order if both succeed. The cuts in Figure 4a indicate `checkpoint=True`.

As an example, we first consider the `acquire` and `commitOrAbort` tasks. `acquire` tasks are nondeterministic because they depend on the run-time state of the external table. `commitOrAbort` has `can_rollback=False` because it is impossible to abort a committed transaction and vice versa. Although `ac-`

quire can be rolled back (e.g., by aborting the transaction and releasing the lock), once we have started the `commitOrAbort` task, it is no longer safe to do so because the transaction may already be committed. Thus, we must ensure that both acquire outputs are saved before `commitOrAbort` starts. We can generalize this rule for the application as follows:

**Invariant 1** (External output commit). *For each workflow task  $v_i$  with `deterministic=False`, let  $G$  be the minimal subgraph that contains  $v_i$  and all downstream tasks (tasks for which there is a path from  $v_i$ ). Then, for each workflow task  $v_j$  with `can_rollback=False` in  $G$ , there must exist a vertex cut that partitions  $v_i$  from  $v_j$  such that all tasks in the cut have `checkpoint=True`.*

Intuitively the vertex cut of the sub-DAG defines a commit point for the nondeterministic output of  $v_i$ . There may exist multiple such cuts. For example, another acceptable specification in Figure 4a is to instead checkpoint the reserve outputs.

ExoFlow guarantees that at least one task frontier is fully checkpointed by the time `commitOrAbort` ( $v_j$ ) starts. Interestingly, this also tells us that we do not need to commit the acquire outputs synchronously. In particular, the reserve tasks in this case are deterministic, as their outputs depend only on whether the lock was acquired and the value stored in the external table, which cannot be modified while locked. Furthermore, their external outputs are not visible while the lock is held. Thus, in this case, it is safe to annotate the reserve tasks with `deterministic=True` and `can_rollback=True`. Together, these annotations allow ExoFlow to *overlap* the checkpoint of acquire’s outputs with execution of the reserve tasks, as long as the checkpoints are synchronized before `commitOrAbort`.

There is a similar requirement for rollback tasks. The rollback tasks in Figure 4a are conditionally invoked by the workflow system to undo external outputs of the acquire tasks. We must ensure that all inputs to the original acquire task are recoverable *before* execution. Otherwise, if the rollback task and its inputs fail simultaneously, it will be impossible to finish rollback. Thus, in Figure 4a, the application must set `checkpoint=True` for `beginTxn`, and ExoFlow synchronizes this checkpoint before executing the acquire tasks.

**Invariant 2** (Rollback durability). *For each path beginning at a task  $v_i$  with `deterministic=False` and ending at a task  $v_j$  that has a rollback function  $R_j$ , there must exist at least one vertex along the path with `checkpoint=True`.*

Unlike Invariant 1, here we only require checkpointing a single task to handle nondeterminism, as the availability of a rollback function  $R_j$  means that we do not need to commit to the original output. The checkpointed task can also be a task other than  $v_i$  or  $v_j$ . For example, if there were additional deterministic tasks between `beginTxn` ( $T$ ) and `rollback_acquire` ( $R$ ), then checkpointing any is sufficient.

Both invariants can be easily checked by walking the DAG passed to run. If an invariant is not met, the system throws an exception to the user. Annotations do therefore require user

```
@ray.remote
class SparkActor:
    def __init__(self):
        self.spark_context = connect(); self.df = None
    def generate_df(self):
        self.df = generate_df(self.spark_context).cache()
    @const
    def exec(self, seed: int) -> int:
        return exec(self.df, seed=seed).count()
    def _checkpoint(self):
        return self.spark_context.save(self.df)
    def _restore(self, path):
        self.df = self.spark_context.load_df(path)
```

Listing 1: Pseudocode for passing a Spark DataFrame by actor. The execution backend implements the actor. Public methods are user-defined. Methods prepended by `_` are called internally by ExoFlow.

cooperation, but note that a user with minimal performance needs can use the defaults in Table 1. This specification trivially satisfies the invariants and indeed corresponds to current workflow systems that commit all task outputs. Section 4 describes how ExoFlow leverages the invariants to improve run-time performance for more sophisticated specifications.

Note that the system will not durably record a nested workflow returned by a task with `checkpoint=False`. To simplify recovery, we disallow sub-tasks with `checkpoint=True`, as we may lose all references to these checkpoints upon failure. We also disallow `can_rollback=False` and `rollback`, as these are challenging to recover without workflow durability.

### 3.4 References

Immutable Refs enable efficient passing of large and distributed data between workflow tasks. For example, Figure 4b shows how the `Ingest` task from Figure 2d can use Refs to return distributed in-memory data. ExoFlow tracks inter-task Ref dependencies for recovery purposes, while the execution backend handles intra-task execution (e.g., `get`).

Some cases require stateful actors for performance. For example, the blue boxes passed between train tasks in Figure 4b are ActorRefs representing a training worker’s state, e.g., a Distributed TensorFlow session. This helps avoid expensive materialization, such as the worker’s local model copy.

Guaranteeing exactly-once semantics for state is challenging. If one task writes the ActorRef’s state, the output is visible to any other task holding a reference to the same actor. This can cause cascading rollbacks on failure depending on how ActorRefs are passed. Furthermore, checkpointing is more challenging if multiple tasks write concurrently to the actor, as the system must ensure that the actor checkpoint is consistent.

To simplify recovery, we limit ActorRef passing to two patterns, analogous to a read-write lock. By default, the ActorRef is in “write” mode. In this mode, only one workflow task may have a reference to the actor at a time. That task can call any actor methods as long as they finish before the task returns. For example, in Figure 4b, only one train task refers

to each actor at a time. ExoFlow can then checkpoint the actors' state between tasks, and on failure, roll back the actors with the workflow. This pattern is useful for abstracting and checkpointing distributed workers in third-party frameworks such as Distributed TensorFlow [9] and Flink [20].

If there are multiple concurrent workflow tasks with a reference to the same actor, however, the tasks are restricted to read-only methods annotated by the user, as shown in Listing 1. Figure 4c shows an expanded Figure 2b in which we use an ActorRef to capture a Spark DataFrame. Initially, A has the only ActorRef, so it can write to the actor's state (`generate_df`). B and C share the actor concurrently, however, and so they are limited to read-only methods (`exec`). Invoking a write method such as `generate_df` would throw a run-time error.

Similar to a read-write lock, ExoFlow can only provide correctness if the application respects certain conditions. In particular, the workflow tasks must explicitly pass ActorRefs through their outputs and arguments. Any other ActorRefs cannot be tracked by ExoFlow and exactly-once semantics is not guaranteed, similar to reading a variable without holding the lock. Also, while methods may be called asynchronously on an ActorRef, a workflow task must synchronize any outstanding calls to an actor before returning.

## 4 Architecture

We describe the ExoFlow design and the requirements of the pluggable execution backend and persistent storage. The workflow controller is a long-running service that can be sharded by workflow (Figure 5). Persistent storage can be implemented by any durable blob storage supporting puts and gets with read-after-write consistency, such as Amazon S3. The execution backend should implement a *remote function invocation* interface, used by the controller to scale checkpointing and task execution. The backend should provide: (1) ability to detect and report task and Ref failures, and (2) guarantee no resource leaks for failed task execution and Refs.

The controller runs as an event loop with the following events: task or checkpoint completes, and task or checkpoint failed. All critical workflow state, such as the workflow DAG, is cached by the workflow controller and written-through to persistent storage, making it simple to also recover the workflow controller. On restart, the controller simply scans the storage for any unfinished workflows, and re-runs to completion.

### 4.1 Workflow execution

The workflow control layer is implemented using the system Ray [37]. Ray provides remote task invocation, distributed immutable memory, and distributed actors. However, Ray only provides at-most-once or at-least-once guarantees and lacks built-in persistence for memory and actors. Thus, Ray tasks and actors are distinct from workflow tasks and actors, which execute exactly-once and can be natively checkpointed.

We use Ray actors to implement the workflow controller and task executors (Figure 5). The controller uses Ray's

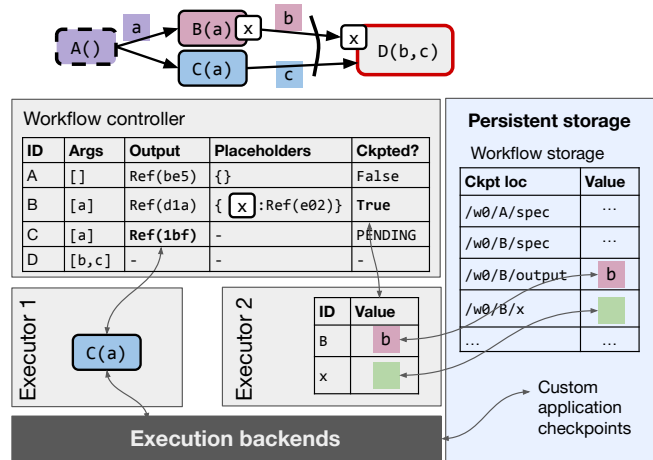


Figure 5: Workflow architecture. The controller and executors are RPC-like services built using Ray actors. Each invocation on these services returns a distributed future (system-internal Refs).

*distributed futures* [47] to coordinate task execution and checkpointing. Distributed futures are an asynchronous extension of RPC where each invocation returns a future pointing to the eventual and possibly remote return value. Ray actors and distributed futures also directly implement application-facing references (Section 3).

We build on Ray for three reasons: (1) futures make it simple for the controller to manage concurrent task execution and checkpointing, (2) passing remote values by reference avoids bottlenecks from large task outputs being passed directly through the centralized controller, and (3) the RPC-like interface straightforwardly and efficiently wraps other execution backends. For example, the Lambdas backend is implemented by wrapping a synchronous Lambda invocation in a Ray task.

The controller is a state machine where the state describes the current execution status of a workflow DAG and is persisted in storage. On run, the controller logs the workflow DAG specification (arguments, opts, etc.) to durable storage and triggers execution. On each iteration of the event loop, the controller may select a workflow task whose inputs are ready and submit the task to an executor. For example, in Figure 5, the controller submits C to executor 1 and immediately receives back the distributed future `Ref(1bf)`. The controller uses this system-internal Ref to wait for task completion, and then passes it to downstream workflow tasks (e.g., D).

Checkpointing is carried out asynchronously by background threads on the executors, enabling parallel and distributed checkpoints that are not bottlenecked by the centralized controller. To checkpoint an output, the executor asynchronously writes a copy to a deterministic storage location (e.g., `w0/B/output` in Figure 5). The controller considers the checkpoint done once it is fully written. For convenience, the controller can also synchronize the checkpoint by requesting a signal from the executor (controller to executor 2 in Figure 5).

Checkpoint synchronization is required: (1) at the end



of a workflow, (2) before executing a task with `can_rollback=False`, and (3) before executing a task with a `rollback` option. Section 6 evaluates a simple policy that synchronizes all pending checkpoints for a workflow in any of these cases and shows that this provides sufficient performance for key applications. A more sophisticated policy may synchronize only the minimum necessary.

ExoFlow handles passing and checkpointing application references (Section 3.4). When a task finishes, the executor replaces any `Refs` and `ActorRefs` appearing in the task’s output with placeholders, e.g., `x` in Figure 5. When passing the output to another task, the controller also passes a list of concrete references (`Ref(e02)` for `x`) used by the executor to fill the placeholders. Task checkpoints include a list of `Ref` checkpoint locations, which are written in parallel and distributed fashion. The controller restores and swaps `Refs` after a failure.

If a workflow task returns a `WorkflowDAG` as its output, the controller simply records the sub-workflow (if `checkpoint=True`), points the output of the parent task to the output of the sub-workflow, then resumes execution.

## 4.2 Workflow recovery

The controller handles task and checkpoint failures. In both cases, the protocol rolls back any previous outputs as needed, then rolls “forward” by re-executing workflow tasks.

The first step is to determine the re-execution task frontier. For example, suppose `C` in Figure 5 fails because we lost `A`’s cached output `Ref(be5)`. Then, we walk the DAG backwards from `C` and add each visited task node to the re-execution set. For each task, we check argument availability, i.e. whether the value has a checkpoint or a live `Ref`. If all arguments are available, then we terminate. Else, we add the tasks that create the arguments (`A`) to the re-execution set. If a visited task has `deterministic=False`, then we also add all tasks downstream to the re-execution set. Thus, if `C` fails and we need to re-execute `A`, we also re-execute `B`, even though it has a checkpoint.

From the re-execution task set, we carry out rollback. In reverse-topological order of the re-execution set, we first clear any cached output `Refs` and output checkpoints, e.g., `/w0/B/output` and `/w0/B/x` for `B`. If it has a `rollback` task, then we re-execute this task, using the same protocol as normal task execution. Finally, we resume workflow execution as normal, starting from the earliest task frontier of the re-execution set.

Critical controller state is persisted, so recovering from controller failure is straightforward. On failure, all in-memory controller state (the table in Figure 5) is wiped, including any `Refs`. On restart, the controller simply scans persistent storage for incomplete workflows, rebuilds its in-memory table, then re-executes them using the described protocol.

**Correctness.** We provide informal proofs that the final outputs are consistent (Definition 1). During normal execution, this follows from the execution protocol: starting from a consistent prefix of outputs, executing a task will produce another consistent prefix.

For recovery, we first consider reconstruction of internal outputs, i.e. values returned by workflow tasks. If the task is deterministic, then the reconstructed output will match the original. If the task is nondeterministic, then the described rollback procedure returns execution to a consistent prefix that does not include any results downstream to the original output.

Next, we consider external outputs: tasks with `can_rollback=False` or `rollback` defined. For a task `T` with `can_rollback=False`, the application guarantees idempotence, so it is enough to show that once `T` begins, the failure-free execution will include the same inputs for `T`. To show this, we rely on Invariant 1 (Section 3.3) and checkpoint synchronization (Section 4.1). The system synchronizes the partition provided by Invariant 1 before submitting `T`; thus once `T` begins, any future recovery procedure will never add `T` to the rollback set.

If `T` instead has `rollback` defined, we must show that if `T` fails, `rollback` will complete with the same view of inputs as `T`’s previous execution, before re-executing `T`. Invariant 2 and checkpoint synchronization guarantee that we can deterministically and idempotently recreate `rollback`’s original inputs.

Correctness also requires preventing conflicts between different executions of the same task. For task checkpoints, if the backend’s failure detection for executors is reliable, then by the time we re-execute `T`, we can be sure that there is no concurrent checkpoint in progress. Under unreliable failure detection, the ExoFlow controller assigns unique checkpoint locations to prevent races between concurrent executions. This requires one extra durable write before each task execution to record the expected checkpoint location.

For a task that returns `Refs` or `ActorRefs`, the execution backend can provide reliable failure detection for references by killing all copies of a `Ref` before reporting failure to ExoFlow. Alternatively, a safe and efficient method that works for both crash and fail-stop failures is to generate unique references for each execution.

## 4.3 Execution backends

**Integration.** ExoFlow references are compatible with existing third-party mechanisms for task communication and recovery. For example, Ray does not provide exactly-once semantics, but it does automatically reconstruct `Refs` created by deterministic (at-least-once) tasks [47]. ExoFlow encourages hierarchical recovery, wherein the execution backend can attempt to handle `Ref` failures first, then throw unrecoverable errors up to the workflow controller.

ExoFlow is compatible with backends that use logging and checkpointing. In general, log-based tasks would use `deterministic=True` and `can_rollback=False` annotations, while checkpoint-based tasks would use `deterministic=False` and `can_rollback=True`. The backend can also directly leverage ExoFlow for checkpointing instead of supplying a user-defined `rollback` function; this shifts the responsibility of checkpoint coordination to ExoFlow and automatically enables optimizations such as overlapping with execution.



**Preventing leaks.** The workflow layer ensures that previous Refs and pending checkpoints do not leak; invalid Refs and checkpoints are dropped during rollback. The execution backend must additionally prevent resource leaks for dead Refs. Dead Refs can be deleted via reference counting (the controller calls back to the backend once a Ref goes out of scope) or garbage collection (the backend scans the controller’s in-memory state for dead Refs).

## 5 Implementation

ExoFlow is built on Ray v2.0.1, which uses gRPC [6] for tasks and actors and a custom shared-memory object store for Ref storage [37]. ExoFlow is implemented as a Ray Python program in 4k LoC.

We implemented two execution backends for ExoFlow: Ray itself (“ExoFlow-Ray”) and the serverless FaaS offering AWS Lambdas (“ExoFlow-Lambdas”). In each case, a typical deployment would use one Ray node to host the ExoFlow controller. In ExoFlow-Lambdas, the controller node takes the place of the gateway provided by AWS for their proprietary serverless workflow offering (Step Functions).

We chose to implement ExoFlow on Ray for three reasons:

1. Support for first-class references to immutable data, which we use to implement Refs.
2. Support for actors (stateful workers), which we use to implement ActorRefs.
3. Low task and actor overhead, similar to pure RPC.

We also use Ray actors to implement executors. Workflow tasks are stateless, but we use actors to store execution state about checkpoints that are pending after task completion.

To build ExoFlow on another actor system such as Akka [2] or Orleans [13], we must implement Refs. This is straightforward for workloads that only pass small data. For data-intensive workflows, one can build a custom in-memory store that is tightly coupled to executors, as in Ray, or use an external key-value store. The latter requires low implementation effort, but may result in poor locality. It is ideal if the execution backend cannot be modified, e.g., to support values larger than the Lambdas response size in ExoFlow-Lambdas.

## 6 Evaluation

Our evaluation covers the following questions:

1. What overheads does ExoFlow add to at-least-once or at-most-once execution backends?
2. How can applications leverage first-class references and task annotations to have greater flexibility in recovery?
3. How does this flexibility in recovery strategy affect performance during execution and recovery?

We compare primarily against these baselines: (1) exactly-once workflow systems: Airflow [3], “standard mode” AWS Step Functions [14], and the serverless workflow system Beldi [49]; and (2) at-least-once distributed DAG systems: “express mode” AWS Step Functions [14] and Ray [37].

Given the high execution overheads of exactly-once workflow systems such as Airflow (Section 6.4), to fairly address questions (1) and (3), we also compare against the following ExoFlow modes:

1. SyncCkpt: Task outputs are synchronized before executing downstream tasks. This is used to simulate the recovery strategy of exactly-once workflow systems such as Airflow.
2. NoCkpt: All task outputs except the final are skipped. This is used to simulate the recovery strategy of an at-least-once or at-most-once system. The application must guarantee that all tasks are deterministic and idempotent to achieve exactly-once semantics.
3. AsyncCkpt: The default mode of ExoFlow. Task outputs are only synchronized where necessary, to provide exactly-once semantics.

We conduct all of the experiments using the AWS cloud, specifically in the us-east-1 region. ExoFlow and execution backends are hosted on EC2 and use Amazon S3 (or EFS in Section 6.2) for persistent storage.

### 6.1 ML training pipelines

We show how ExoFlow enables a flexible recovery-performance tradeoffs for the workflow in Figure 2d. We use an image classification example adapted from Azure MLOps [4]. An ETL Ingest task (1 r3.2xlarge node) downloads the compressed data from S3. “1×” in Figure 6 indicates one data copy with 569 raw image files and total size 225MB. The task loads the images into memory, and performs data cleaning and normalization with at-least-once parallel Ray tasks. The dataset (1.4GB of memory per data copy) is partitioned and passed using Refs to the dataset augmentation tasks, via Ray’s shared-memory object store. Dataset augmentation again uses Ray at-least-once tasks to apply random cropping, flipping, and color adjustments to the base dataset, once per epoch. Dataset augmentation requires repeatedly processing the same dataset in a tight loop with training. Therefore, the dataset augmentation stage accumulates a total intermediate and checkpoint size of 67GB and 18GB respectively, per data copy. Training tasks are colocated and pipelined with dataset augmentation (1 g4dn.12xlarge node, 4 NVIDIA T4 GPUs). We use PyTorch data-parallel distributed training and the ConvNeXt Tiny (28.6M parameters) model. PyTorch workers are passed using ActorRefs.

Figure 6L shows end-to-end duration of 25 epochs without failures of different ExoFlow recovery modes, as a function of dataset size. Here, we also include Selective AsyncCkpt (skip checkpointing dataset augmentation outputs) and Workflow Tasks (include at-least-once Ray tasks for data processing in the workflow DAG instead of passing volatile Refs).

Duration predictably grows approximately linearly with the dataset size for all strategies. The overhead of Workflow Tasks is high because each data processing task is durably (and unnecessarily) logged as part of the workflow. For the same

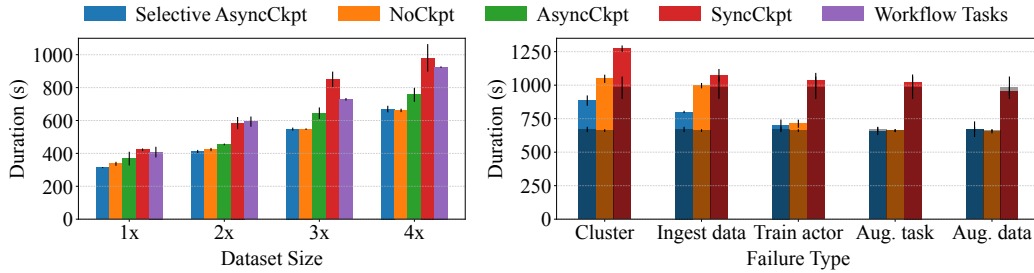


Figure 6: End-to-end duration for the ML workflow application shown in Figures 2d and 4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure.

workflow graph, the overhead for larger data varies depending on the recovery strategy. NoCkpt represents the best possible performance, where only the final model is checkpointed. SyncCkpt represents existing workflow systems (Figure 2c) and its overhead grows the most because checkpointing overhead grows faster than computation overhead. AsyncCkpt’s overhead grows less because checkpointing of augmented datasets is overlapped with training tasks. Selective AsyncCkpt has nearly identical duration as NoCkpt because the Ingest checkpoint is perfectly overlapped with training tasks.

Meanwhile, Figure 6R shows end-to-end duration in different failure scenarios compared to normal run-time execution (dark): whole cluster failure (including the ExoFlow controller); in-memory ingest data lost; PyTorch worker actor lost; augmentation task lost; and in-memory augmented data lost. Here, we see the tradeoff between recovery and performance. SyncCkpt has similar or better recovery time overhead than NoCkpt for cluster and ingest data failures because it avoids re-executing the Ingest task, but overall it does worse because of high normal run-time overhead. Selective AsyncCkpt checkpoints the Ingest data asynchronously, so recovering from cluster and ingest data failures is fast because it simply restores the Refs from the checkpoint. Together, Figure 6L and R demonstrate how *the developer can flexibly choose the best recovery strategy*.

Figure 6R also demonstrates ExoFlow’s *broad failure coverage and ability to integrate with Ray’s built-in recovery*: Ray automatically reconstructs deterministic data processing results but does not handle persistence or actor recovery [47]. Thus, ExoFlow handles the first four failures, while Ray handles the last. Recovery for the last two failures is fast because rollback and checkpoint restore are unnecessary.

## 6.2 Stateful serverless workflows

We compare ExoFlow on a travel reservation benchmark [26] to Beldi [49], a recent system for fault-tolerant and transactional stateful serverless workflows that uses *intent logging* to ensure exactly-once semantics. Our implementation uses Beldi’s APIs for reading and writing state but the ExoFlow controller with an AWS Lambdas backend for workflow execution and recovery. We use a single m5.16xlarge instance to host ExoFlow and EFS for persistent storage, which

provides lower latency than S3. The benchmark procedure follows [49], and we report response latency in Figure 7a.

ExoFlow achieves about 51% lower p50 latency than Beldi for request rates up to 400, despite using the same execution system (AWS Lambdas) and state APIs (Beldi). This is because most of the workflows have deterministic computation and no external effects (i.e. read-only), so the additional logging used by Beldi is unnecessary for correctness. Furthermore, Beldi schedules an additional Lambda function to orchestrate others, while ExoFlow directly schedules Lambdas. When requests/s is higher than 700, ExoFlow’s median latency is greater than Beldi’s. This is due to the Lambdas invocation bottleneck at the ExoFlow controller node and can be easily removed through sharding across workflows. The Lambdas gateway used in Beldi is likely sharded internally.

The use of ExoFlow as a Lambdas gateway has benefits in recovery time. Figure 7a also shows latency with a 10% failure rate for all Lambdas. ExoFlow directly invokes Lambdas, so it can detect failures and recover virtually instantaneously, resulting in 0-31% extra overhead in p99 latency. In contrast, Beldi is fully decentralized and relies on timeouts for recovery correctness. Thus, although Beldi-style logging may reduce re-execution on recovery, the actual recovery time would be lower-bounded by a timeout ([49] evaluates 1min as a possible lower bound).

Figure 7b further demonstrates the performance benefit of exposing application semantics to the workflow system. We report latency of the most complex workflow in the benchmark, the trip reservation request described in Figure 3. Beldi implements the transaction using two-phase locking (2PL). We demonstrate progressive improvement over the original solution by varying the execution and recovery strategy. First, we eliminate Beldi logs for dynamic task invocation, as the DAG can be easily specified upfront, reducing p50 and p99 latency by 17% and 25% respectively (-wal). Next, we parallelize the hotel and flight reservation tasks, further reducing p50 and p99 latency by 17% and 15% respectively (+parallel). Beldi executes these tasks sequentially because asynchronous invocation does not allow retrieval of the reply. Finally, we split each reservation task into two steps: lock acquisition and reservation, as seen in Figure 4a. -async shows that with synchronous checkpoints,

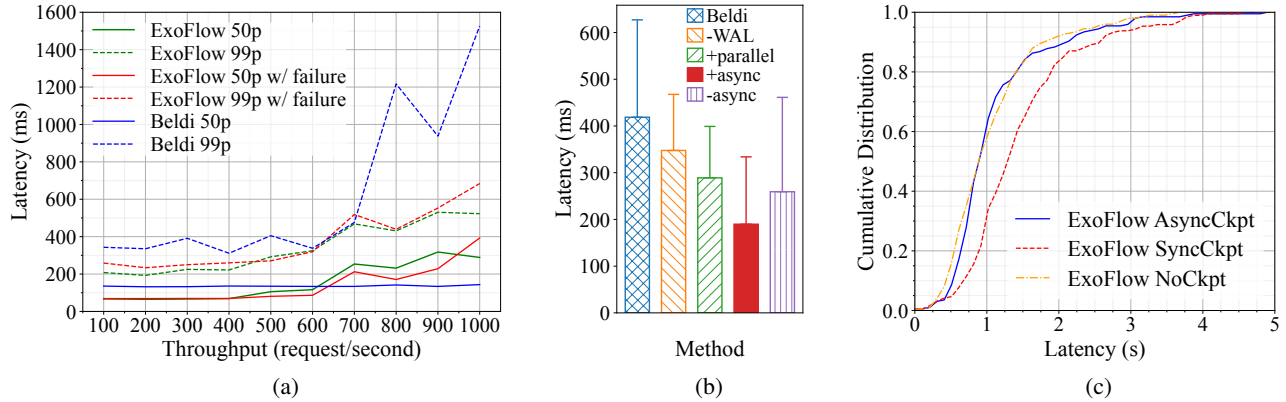


Figure 7: (a) Response latency percentile for a serverless travel reservation benchmark [25]. (b) Median latency of the trip reservation request from the travel reservation benchmark. Error bar represents 99-percentile latency. (c) Latency CDF of online-offline graph processing.

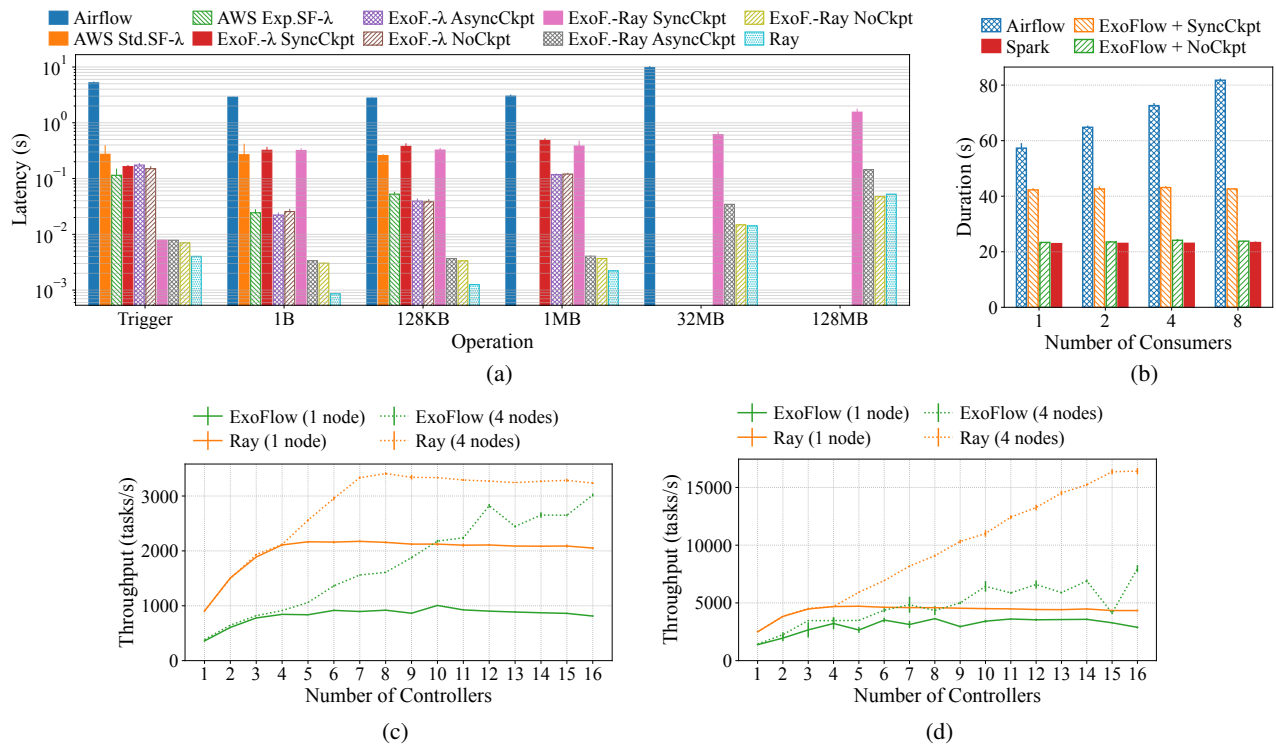


Figure 8: Microbenchmarks. (a) Triggering and data passing latency of ExoFlow and other workflow systems, using AWS Lambda ( $\lambda$ ) and Ray as execution backends. Missing bars indicate limitations in inter-task communication. (b) End-to-end run time for the ETL workflow shown in Figures 2b and 4c, compared with Airflow and native Spark. (c, d) Maximum task throughput (c: 1 task/DAG; d: 100 tasks/DAG) of 10k tasks, compared against Ray as an optimal baseline, on 1 node and 4 nodes.

this actually increases latency due to the added task. However, +async shows that by overlapping checkpointing with execution, we can further reduce p50 and p99 latency by 34% and 16% respectively, without compromising correctness.

### 6.3 Online-offline graph processing

Distributed graph processing systems can be generally divided into stream vs. batch processing [36]. Streaming systems can handle continuous updates and produce timely results, but may not offer the same precision as batch systems.

We use references in ExoFlow to link stream and batch

graph processing, producing a single application that can both handle online queries and produce periodic exact results. We use Ray actors to implement a version of Kineograph [21], a streaming graph processing system that uses distributed snapshots for consistency. Each workflow task ingests one epoch of incoming graph updates to compute a graph snapshot and an online approximate result, and we periodically pass the snapshot in-memory to another workflow task that uses Spark to compute the full result.

We evaluate on the SNAP Twitter follower network

dataset [33] (41M nodes and 1.5B edges), with each input record representing an edge insert event. We run the push-model TunkRank algorithm used by Kineograph to compute Twitter user influences on a 3-node r3.8xlarge cluster, 1 for streaming and 2 for the Spark cluster. We use two Ray actors to process the input stream and use ExoFlow to checkpoint and pass the ActorRefs between streaming tasks. Each streaming task represents a 10-second epoch and also returns 4 Refs that represent the partitioned graph snapshot. These Refs are passed to a Spark task every 20 epochs. Latency is reported for 200 epochs, after an initial warmup of 150 epochs. The average digestion rate is 44.94k tweets per second with our dataset. Kineograph achieves about 40k tweets per second with 2 ingest node + 48 graph nodes with a similar setting. We outperform Kineograph likely because we utilize shared memory for data passing, with more powerful hardware, which significantly reduces overhead of data pushing.

Figure 7c shows a CDF for latency from input event to the earliest time that the event is reflected in a streaming task’s output (although inconsistent results can be returned earlier by querying the ingest actors directly). AsyncCkpt allows the snapshot to be viewed before it is checkpointed. NoCkpt has impractical recovery overhead, but we use it as a performance baseline. AsyncCkpt achieves similar latency as NoCkpt, meaning that checkpointing overhead remains stable as the graph grows larger; this is because streaming tasks pass through previous Refs that are already checkpointed, so ExoFlow only checkpoints new data on each epoch. SyncCkpt is similar to Kineograph, checkpointing the snapshot before making it visible, and adds less than 1s latency. Finally, the error rate of the online results and the batch processing task duration both grow linearly over time, confirming the tradeoffs between batch vs. stream processing.

## 6.4 Microbenchmarks

**Latency.** With equivalent backends, ExoFlow matches or reduces execution overheads of existing workflow systems while enabling more flexible inter-task communication. Figure 8a (1 m5.8xlarge instance) shows the latency of workflow execution (“Trigger”) and task execution with different size arguments. We use exactly-once systems (Airflow [3], AWS Standard Step Function [14]) and at-least-once systems (AWS Express Step Function [14], Ray [37]) as baselines. Airflow is an industrial custom-built workflow system while Step Functions are the AWS-native workflow offering for Lambdas.

First, with the Lambdas backend, ExoFlow has similar trigger latency as AWS Standard Step Function. Airflow has generally high overhead due to coordinating execution through a database, which can easily lead to inefficient scans.

“1B” in Figure 8a compares minimum task execution latency. ExoFlow-Lambdas achieves comparable latency as AWS Step Functions, as the primary overheads for exactly-once and at-least-once execution come from durability and Lambdas invocation, respectively. ExoFlow-Ray improves

upon the latter as it uses Ray for execution.

Finally, we compare the ability to pass large data between tasks. AWS Step Functions limit data passing to 256KB, but plain Lambdas have a size limit of 6MB. Thus, ExoFlow-Lambdas can actually support larger data sizes. This could be improved further with Refs, e.g., with Redis [44] for distributed memory. Airflow’s XCom [1] can support slightly larger data but is fundamentally limited by its database-centric design. Meanwhile, ExoFlow-Ray uses Ray Refs for efficient data passing. The gap between AsyncCkpt and NoCkpt latency is small but grows with data size; although the checkpoint is asynchronous, ExoFlow synchronously copies the data to guard against concurrent writes.

In summary, ExoFlow’s low execution overheads make it a practical replacement for existing workflow systems, and it enables greater flexibility in task communication and recovery.

**Data sharing for ETL.** We evaluate ExoFlow against Airflow for a Spark workflow similar to Figure 2b (1 m5.8xlarge instance, 4GB Spark memory). Figure 8b measures total run time for a workflow that uses Spark to generate a 1GB random dataset, followed by multiple downstream tasks that consume the data with data sampling Spark jobs. Such a workflow requires orchestration across Spark jobs, which Spark does not provide, and is therefore often run on a workflow orchestrator such as Airflow.

Airflow run time grows proportionately with the number of consumers because they cannot share data in-memory. Meanwhile, ExoFlow scales well even with synchronous checkpointing because consumers share data via Spark’s native cache. Furthermore, ExoFlow runs as fast as native Spark alone, while facilitating composition with other systems as well.

**Throughput and Scalability.** We measure maximum throughput with varying numbers of controllers, (AWS m5.2xlarge) nodes, and tasks per DAG. We use Ray as the optimal baseline, as Ray is also the execution backend.

Figure 8c (1 task/DAG) shows that ExoFlow and Ray both reach saturation after 4 controllers on one node. With 4 nodes, scalability continues, and the gap between ExoFlow and Ray narrows at around 16 controllers. Figure 8d (100 parallel tasks/DAG) shows that throughput overall improves via task batching. Again, with four nodes, both ExoFlow and Ray scale linearly with the number of controllers. ExoFlow achieved roughly 50% of Ray’s throughput, due to additional overhead from workflow orchestration and ensuring exactly-once semantics.

## 7 Related Work

**Workflow systems.** Industry workflow systems [3, 5, 7, 14] orchestrate execution and recovery for distributed applications by durably logging the workflow, checkpointing task outputs and replaying failed tasks. However, they require external outputs to be idempotent and significantly limit how tasks can pass data to each other (Section 2).



Many workflow systems for FaaS focus on stateful serverless workflows. Several provide a fault-tolerant transactional key-value store interface [45, 46, 49]. ExoFlow is agnostic to external state APIs and implementation and factors out execution and recovery orchestration from such systems.

Some stateful workflow systems offer a fault-tolerant actor programming model [8, 15, 16]. A common recovery technique is *event sourcing*, i.e. durably logging nondeterministic events. However, this requires the developer to use special APIs for nondeterministic code and can add higher overheads than necessary when deterministic replay is not required for application correctness [23, 38]. ExoFlow also supports pluggable actors but only with coarse-grained logging (i.e. recording the workflow DAG) and checkpoint-based recovery (Section 3.4). This is intentionally minimal, as it enables composition of both log- and checkpoint-based actor implementations.

ExoFlow is similar to DARQ [35]: both use composable atomic steps (tasks) and asynchronous checkpointing. Unlike DARQ, ExoFlow exposes references and annotations to avoid materializing and/or persisting outputs where possible.

**Dataflow systems.** Many dataflow systems use the DAG model [22, 31, 48]. Several use *lineage reconstruction* for recovery, a form of logging that records the DAG but not the data, to reduce run-time overhead. CIEL [39, 41] also introduces *dynamic tasks*, which we adopt. However, these systems target data processing applications in which all tasks are stateless and deterministic. Ray proposes a unified API for DAGs and actors [37], which we also adopt, but cannot support exactly-once semantics or persistence [47]. Tachyon [34] proposes a method of optimizing checkpoints for lineage-based systems; this could be applied to a future version of ExoFlow.

Other systems such as Naiad [38], Apache Flink [18] and Canary [43] implement both batch and streaming dataflow with message passing and global checkpoints at run time for recovery. This produces lower latency but requires more rollback on failure; it can also add more overhead for applications with frequent external outputs [23]. ExoFlow augments log- and checkpoint-based systems by orchestrating recovery across systems with different internal strategies (Section 6.3).

Falkirk Wheel [27] targets efficient and flexible recovery for batch *and* streaming. It uses logical message timestamps to transparently determine the minimum to roll back on failure. ExoFlow provides practical recovery for black-box functions (tasks) by asking semantics from the developer through references and task annotations.

**Actor systems.** The actor model is a distributed programming model where processes communicate through asynchronous method calls [30]. Most systems do not guarantee exactly-once semantics [2, 12, 17, 47]. ExoFlow provides a limited exactly-once actor model to support workflows that pass actors between tasks. Meanwhile, the application has full flexibility of existing actor systems within a task.

**Message-passing systems.** Message-passing systems are a generalization of actors in which processes communicate

through message sends and receives. There is a large body of work on recovery for message passing, primarily focusing on logging vs. checkpointing [23]. Our work adapts these techniques to the distributed workflow setting and aims to compose log- and checkpoint-based applications.

## 8 Discussion

**References for framework interoperability.** Like other dataflow systems, ExoFlow captures the *logical* data movement in an application. ExoFlow also aims to enable *interoperability* across distributed execution frameworks, unlike abstractions such as RDDs [48] or timely dataflow [38] that are tightly coupled to a specific framework. This motivates some of the differences between Refs and ActorRefs vs. other dataflow abstractions: they can be used to capture third-party data and context, they are serializable, and they do not impose a particular model of parallelism.

**Limitations.** Using ExoFlow effectively requires developer effort. ExoFlow offers recovery flexibility but the developer must choose the right tradeoff for their application. For example, the developer must decide how large a workflow task should be, and whether checkpointing the output is desirable. Currently task annotations are also very coarse-grained, which makes the system general-purpose but also makes it more challenging for an application to achieve optimal performance and recovery overheads.

There are a number of future directions towards improving ExoFlow's interfacing with external systems. First, while Refs allow the application to efficiently pass data between workflow tasks, reading and writing a Ref's data may still require data movement to or from an external framework. Second, currently ExoFlow does not support transactions, i.e. there is no way to specify that a task should be rolled back if another task fails. In this case, the developer must manually roll back the effects of both tasks, e.g., in a final `commitOrAbort` task. Finally, for cases where tasks read and write external state, capturing more fine-grained semantics could reduce developer burden and improve performance. For example, native support for popular types of external state (e.g., a database) could be added.

## 9 Conclusion

Many existing distributed systems provide specialized, efficient, and transparent recovery for specific application domains. ExoFlow has an orthogonal and complementary goal. To unify heterogeneous applications, we must provide *general* and *interoperable* recovery methods. The greatest challenge is to gain sufficient application semantics without sacrificing flexibility. ExoFlow presents one approach that strikes a balance between usability (minimal annotations, compile-time safety checks) and functionality (flexible Refs, automatic recovery). In doing so, we hope to provide universal recovery that matches a universal API: the workflow DAG.

## Acknowledgement

We thank the OSDI reviewers and our shepherd, Steven Hand, for their valuable feedback. We also thank Haoran Zhang and Vincent Liu for their insightful discussions and help with Beldi. This work is in part supported by NSF CISE Expeditions Award CCF1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware.

## References

- [1] Airflow XComs. <https://airflow.apache.org/docs/apache-airflow/stable/concepts/xcoms.html>. Accessed: 2022-12-13.
- [2] Akka. <https://akka.io/>.
- [3] Apache Airflow. <https://airflow.apache.org/>.
- [4] End-to-end mlops pipeline example on azure. <https://github.com/microsoft/MLOps/tree/master/examples/KubeflowPipeline>.
- [5] Google Cloud Composer. <https://cloud.google.com/composer>.
- [6] gRPC. <https://grpc.io>.
- [7] Kubeflow. <https://www.kubeflow.org/>.
- [8] Temporal. <https://temporal.io/>.
- [9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, Georgia, USA, 2016*.
- [10] Michael Armbrust. SPARK-20928: Continuous Processing Mode for Structured Streaming. <https://issues.apache.org/jira/browse/SPARK-20928>, 2017.
- [11] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.
- [12] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.
- [13] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.
- [14] Jyothi Prasad Buddha and Reshma Beesetty. Step functions. In *The Definitive Guide to AWS Application Integration*, pages 263–342. Springer, 2019.
- [15] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.
- [16] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.
- [17] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.
- [18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.
- [19] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.
- [20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.
- [22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [23] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
- [24] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia,

- and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019.
- [25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.
- [26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.
- [27] Ionel Gog, Michael Isard, and Martín Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 373–387, 2021.
- [28] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.
- [29] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [30] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI’73*, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys ’07*, pages 59–72, New York, NY, USA, 2007. ACM.
- [32] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.
- [33] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [34] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15, 2014.
- [35] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. In *Proceedings of the ACM on Management of Data*, 2023.
- [36] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.
- [37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elilbol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.
- [38] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, pages 439–455, New York, NY, USA, 2013. ACM.
- [39] Derek G. Murray, Malte Schwarzkopf, Christopher Snowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI’11*, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.
- [40] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.
- [41] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.
- [42] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, 2019.

- [43] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.
- [44] Salvatore Sanfilippo. Redis: An open source, in-memory data structure store. <https://redis.io/>, 2009.
- [45] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.
- [46] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
- [47] Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *NSDI*, pages 671–686, 2021.
- [48] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.
- [49] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.



## A Artifact Appendix

### Abstract

Our artifact includes a comprehensive guide and the source code of the project that allows the evaluators to validate the claims made in ExoFlow. Our artifact runs on Amazon AWS without additional requirements or dependencies. Deploying the code, performing the measurements, generating the plots, and running the benchmarks depend on some third-party frameworks including Anaconda, awscli, and Ray. Please refer to our Github repository <https://github.com/suquark/ExoFlow> for the latest instructions on reproducing the results.

### Scope

The artifact allows the evaluators to validate the claims made in the ExoFlow paper (mostly in the figures) and provides a means to replicate the experiments described. The artifact can be used to set up the necessary environment, execute the main results, and perform microbenchmarks, thus providing a comprehensive understanding of ExoFlow's capabilities.

### Contents

Our artifact includes a comprehensive guide designed to assist the evaluator in setting up and running experiments for the ExoFlow paper. It is organized into three primary sections: Local Setup, Main Results, and Microbenchmarks.

The Local Setup section provides instructions to set up an initial AWS EC2 instance. All subsequent experiments will be conducted within that instance.

The Main Results section contains instructions to reproduce the main experiments (ML training pipelines, Stateful serverless workflows, Online-offline graph processing) presented in our paper. These experiments may take a significant amount of time to run (>30 hours) for evaluation. Therefore, we provide options for both batch running experiments and testing individual data points.

The Microbenchmarks section includes instructions for running microbenchmarks, which take a shorter time to complete.

### Hosting

You can obtain our artifacts from GitHub: <https://github.com/suquark/ExoFlow>. The Github repository may be updated later, but we will maintain clear and accessible instructions about our artifacts in an easily identifiable "README" file.

### Requirements

ExoFlow is developed and tested on AWS, and we use some AWS services as the baseline. Thus, an AWS account and quota for certain experiments are required.

# Hyrax: Fail-in-Place Server Operation in Cloud Platforms

Jialun Lyu<sup>1,2</sup> Marisa You<sup>1</sup> Celine Irvine<sup>1</sup> Mark Jung<sup>1</sup> Tyler Narmore<sup>1</sup> Jacob Shapiro<sup>1</sup>  
Luke Marshall<sup>1</sup> Savyasachi Samal<sup>1</sup> Ioannis Manousakis\* Lisa Hsu\* Preetha Subbarayalu<sup>1</sup>  
Ashish Raniwala<sup>1</sup> Brijesh Warriar<sup>1</sup> Ricardo Bianchini<sup>1</sup> Bianca Schroeder<sup>2</sup> Daniel S. Berger<sup>1,3</sup>  
<sup>1</sup>Microsoft Azure    <sup>2</sup>University of Toronto    <sup>3</sup>University of Washington

## Abstract

Today’s cloud platforms handle server hardware failures by shutting down the affected server and only turning it back online once it has been repaired by a technician. At cloud scale, this all-or-nothing operating model is becoming increasingly unsustainable. This model is also at odds with technology trends, such as the need for new cooling technology.

This paper introduces Hyrax, a datacenter stack that enables compute servers with failed components to continue hosting VMs while hiding the underlying degraded capacity and performance. A key enabler of Hyrax is a novel model of changes in memory interleaving when deactivating faulty memory modules. Experiments on cloud production servers show that Hyrax overcomes common hardware failures without impacting peak VM performance. In large-scale simulations with production traces, Hyrax reduces server repair requirements by 50-60% without impacting VM scheduling.

## 1 Introduction

Server hardware failures are quite frequent in cloud platforms. For example, a typical cloud server relies on at least 24 DIMMs, six SSDs, six fans, and two CPU sockets [48]. Even assuming optimistic annual failure rates<sup>1</sup> of 0.1% per DIMM and 0.2% per SSD, 22% of servers will have at least one failure during the typical 6-year lifetime of a cluster. In practice, repair rates are typically even higher.

The common approach to dealing with hardware failures in today’s cloud platforms is to evict all virtual machines (VMs) and stop using the affected server. The server goes back into production only once a technician has replaced all faulty components. This maintains server homogeneity, which simplifies scheduling and operation [4, 24, 32, 41, 42, 46, 61, 64, 69]. We call this the “all-or-nothing” operating model.

Recent technology trends make all-or-nothing operations increasingly unsustainable in cloud platforms. First, server power consumption increasingly requires liquid cooling, which offers performance, efficiency, and sustainability benefits [33, 63, 72]. Liquid cooling significantly increases the time and effort required to repair servers. Second, the share

of total costs that are due to repairs are increasing (§2). This is in part due to servers staying in datacenters for longer<sup>2</sup>. Third, all-or-nothing requires a continuous supply of spare components, which is increasingly hard to procure. Component supply chains have emerged as a barrier to further extending server lifetimes and reducing carbon emissions [7]. Fourth, the human repair process can cause interruptions to nearby servers [32], which is becoming an obstacle in cloud provider’s pursuit to improve the availability of their servers.

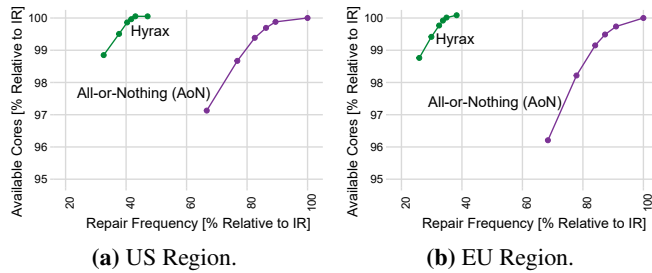
This paper advocates that cloud providers should move toward a fail-in-place paradigm where servers with faulted components continue to host VMs without requiring repairs. Fail-in-place operation would significantly reduce repair needs, improving costs, carbon emissions, and availability. However, fail-in-place faces multidimensional challenges in practice. First, it requires a form of graceful degradation where individual faulty components are deactivated instead of decommissioning the entire server. Unfortunately, we find that mechanisms to deactivate components are largely undocumented. Furthermore, deactivating the right component requires accurate fault diagnostics and it is unclear whether this can be achieved in practice. Second, deactivating common components such as DIMMs can significantly impact server performance due to reduced memory interleaving. This performance loss should not be exposed to VM customers. Third, the cloud platform must be able to actually use the capacity on servers with deactivated components. This requires algorithmic changes in VM scheduling and changes to adopt the cloud control plane to support heterogeneous servers.

We introduce Hyrax—the first implementation of the fail-in-place paradigm for cloud compute servers. In a multi-year study of component failures across five server generations, we find that sufficient redundancy in existing servers can overcome the most common memory and SSD device failures. While existing diagnostics can only identify a subset of component types, we empirically find that they are 95% accurate. We identify hooks in deployed firmware that enable deactivating components in ways that overcome many failure possibilities (e.g., dirty or corroded connectors or chip failures). Finally, Hyrax adds a *degraded* server state and corresponding scheduling rules to a production control plane to

\*Formerly at Microsoft Azure

<sup>1</sup>Prior work reported 0.09% [58, 59], 0.12% [12], and 1.6% [57, 66] for DIMMs and 0.22% [43, 44] to 1.2% [3, 54] for SSDs.

<sup>2</sup>Major cloud providers have moved to a minimum server lifetime of six years [7, 26, 53] for cost and sustainability reasons.



**Figure 1:** Hyrax dominates all-or-nothing (AoN) operations along the entire trade-off spectrum between available resources (core hours) and the number of required server repairs (repair tickets). Different points on the trade-off spectrum are generated by varying the repair schedule, ranging from immediate repairs (IR) to performing repairs in batches at periodic intervals ranging from 1-12 months long. All numbers are normalized to those for AoN with immediate repairs, which is the common approach in today’s cloud platforms.

support servers with deactivated components.

Hyrax overcomes the reduced performance of degraded servers by exploiting existing heterogeneity in VM sizes and configurations. Specifically, we find that the peak performance expectation of small and old VM types matches the performance offered by degraded servers. Further, we find that there are sufficiently many small and old VM types to effectively utilize the capacity of degraded servers. Hyrax also introduces scheduling optimizations for efficiency at scale.

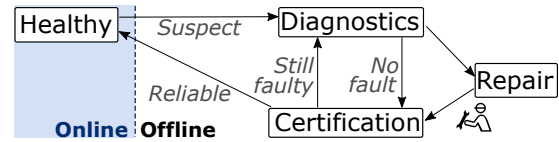
Hyrax has been deployed for a few months on a subset of Azure clusters and a small set of component types. We report on its effectiveness on real failures and use microbenchmarks and large-scale trace-driven simulations to extrapolate a full deployment over six years. Our experience demonstrates that the fail-in-place paradigm is practical under real-world platform constraints.

To evaluate the benefits of at-scale deployment, we simulate 66 compute clusters from two geographic regions over a period of six years. Overall, Hyrax reduces the number of server repairs in a region by 50-60% depending on the region (Figure 1), while offering the same resource availability and scheduling the same VMs as today’s all-or-nothing operation. Figure 1 also shows that Hyrax’s benefits carry over to different repair schedules, including Azure’s existing repair schedule (immediate repairs) as well as previously-suggested batching of repairs [4, 5], where repairs are scheduled at periodic intervals (e.g. once per year). Furthermore, Hyrax reduces replacement rates by 40% for fans, 50% for SSDs, and 75% for memory, which enables extending server lifetimes for multiple years to amortize server costs and carbon emissions.

We hope that, by sharing our journey towards the fail-in-place paradigm, we motivate the community to invest in future cross-stack systems research to make degraded mode and fail-in-place operation significantly more efficient.

**Contributions:**

- The first description of design goals and constraints for



**Figure 2:** At Azure, servers are either online and serving VMs, or offline and being repaired. Repairs take between 3 and 190 days at the 50-th and 99-th percentile, respectively.

fail-in-place and feasibility analysis of degraded mode operation at a large public cloud platform (§3).

- The design and implementation of Hyrax, the first fail-in-place system at a cloud provider. Hyrax’s implementation includes novel mechanisms to deactivate component pathways and a novel model of memory interleaving when memory modules are deactivated (§4, §5, and §6).
- Experimental results that show Hyrax’s effectiveness, performance, and cost impacts (§7).
- A discussion of deployment experience, broader impacts, and research avenues (§9).

**Limitations.** Hyrax is not applicable to all repair operations. The following assumptions underpin our work.

- Hyrax focuses on server repairs, which account for the majority of technician hours in Azure datacenters. Hyrax does not reduce other technician duties, such as power, network, and cooling maintenance.
- Hyrax focuses on compute servers, where degraded operation is challenging. Storage servers often already implement variants of degraded mode (§8).

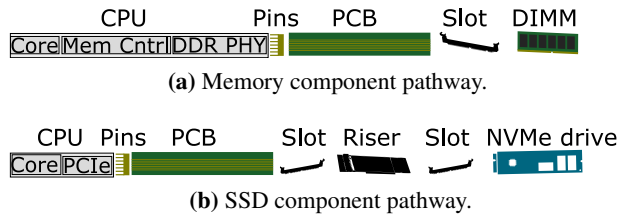
**2 Background**

This section reviews repair workflows and costs, typical server configurations, and cloud workloads.

**Repair workflow.** A software agent called Server Health Monitor (SHM) checks server error logs and component types, counts, and capacity for deviations from the expected (homogeneous) configuration. If the SHM suspects any kind of fault, the server is marked as “offline”, which signals the VM scheduler to filter out this server (Figure 2). VMs are migrated away or gracefully evicted. The server is then rebooted into a diagnostics environment. If diagnostics finds a hardware problem, it immediately creates a *repair ticket* [4, 32, 41, 66].

Repair tickets can point to a specific component pathway (like DIMM #4, Figure 3a) or require a manual diagnosis. After a technician resolves a ticket, e.g., by reseating connectors or swapping out components, the server is tested again to certify reliability (certification step). A reliable server is marked “online” and again becomes a candidate for hosting VMs.

**Impact of all-or-nothing repairs on TCO.** Server repairs are a significant component of total cost of ownership (TCO). The main components of TCO are CapEx (capital expendi-



**Figure 3:** A component can appear faulty due to other component faults along the path between a core and the actual component. We call this a *pathway* that typically spans the socket and pins, printed circuit board (PCB), and slots/risers to the actual component like the NVMe SSD or memory DIMM.

ture for the purchase of servers, networking, cooling, and power infrastructure) and operational costs due to energy and power (estimated at 6% of CapEx per year [19, 20, 62]), and maintenance (estimated at 5% of CapEx per year for each server [4, 66]). Maintenance costs are largely made up by technician salaries and cover maintenance of all datacenter components. At Azure, server repairs account for about half of technician work hours in the all-or-nothing operating model. Server repairs thus account for 9% and 12% of total cost (TCO) for server lifetimes of 6 and 10 years [7], respectively.<sup>3</sup>

Repairs are also known to be slow [69]. At Azure, 2% of servers are waiting for repairs at any given time in the all-or-nothing operating model.

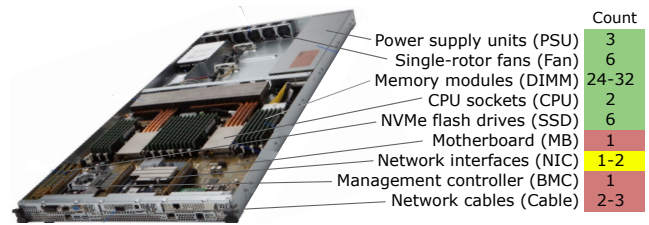
**Server hardware.** Figure 4 shows a typical cloud server configuration [48]. Variants of this base architecture include one or two NICs and 24-32 DIMMs; most servers use a single NIC. We note that the component count for some component types is larger than one (marked in green in Figure 4). We refer to these as *degradable* components as they do not represent a single point of failure.

We note that hardware components internally contain redundancy, such as spare blocks in SSDs [8, 25, 34, 44, 52, 55, 67, 68]. Moreover, the operating system and hypervisor at Azure employ an aggressive policy for offlining memory pages to mask faulty cachelines. A repair ticket is generated for a component only when the above mechanisms cannot resolve the problem.

**Cloud workload.** All workloads run within virtual machines (VMs) for security and ease of management. Resources for each VM are typically preallocated at its start time to improve performance and facilitate the use of virtualization accelerators [2, 39, 60, 70, 71]. VMs come in hundreds of different types with many combinations of the number of virtual cores, memory capacity, local and remote storage options, NIC and GPU configurations.

The cloud provider has no introspection into the workloads that a customer is running inside their VMs and does not know their performance requirements. Hence, performance goals

<sup>3</sup>We calculate TCO based on the three dominant cost factors: Deployment years ( $y$ ), CapEx ( $C$ ), Maintenance ( $y \times C \times 5\%$ ), and Energy/Power ( $y \times C \times 6\%$ ). This leads to  $TCO(y) = C + y \times 5\% \times C + y \times 6\% \times C = C(1 + 0.11 \times y)$ .



**Figure 4:** A typical cloud server configuration and its component counts. We refer to the component types marked in green as *degradable*, as their component count is large enough that they are not a single point of failure.

are defined in terms of peak performance, e.g., bandwidth and latency for memory and IOPS and bandwidth for SSDs. For older VM types that are scheduled on newer servers, their performance goals are defined for the server generation they were originally introduced on.

Azure’s distributed VM scheduler is called Protean [1, 9, 22, 37, 61]. Protean first forwards VM requests to a compute cluster within the specified region based on hardware requirements and available capacity. At the cluster level, Protean places VMs following a series of rules that balance tightly packing resources with spreading workloads across racks for high availability. Filter rules select which servers are considered candidates for placing each VM. They ensure that only servers are considered that can ensure the SLAs associated with the requested VM type. Preference rules rank these candidates to find the best placement. Similar to other schedulers [4, 22, 24, 32, 41, 42, 46, 61, 64, 69], Protean assumes identical hardware configurations for all servers within a cluster.

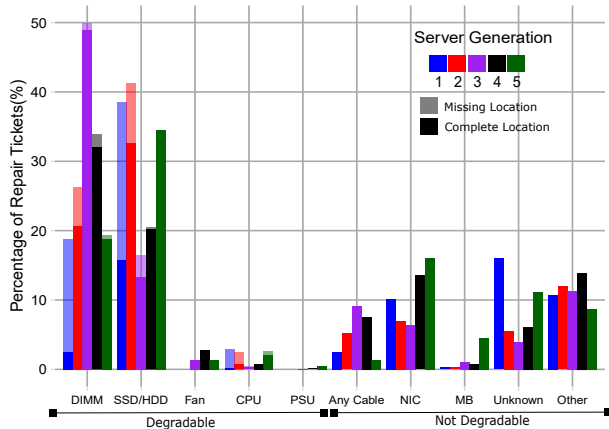
### 3 Fail in Place

The “all-or-nothing” operating model and the associated high repair frequency is costly and at odds with multiple server and data center trends. This paper pursues an alternative paradigm, which we term Fail-in-Place (FIP). In FIP, servers are allowed to exist with failed components for prolonged periods of time, sometimes forever. The main goal of FIP is to *reduce repair tickets* while continuing to offer the same user experience to VMs and minimal impact on cluster capacity and scheduling.

FIP is motivated by our observation that the majority of hardware repair tickets are due to the failure of *degradable* components. Consider Figure 5, which breaks down repair tickets at Azure into the component type that triggered them. We see that, for example, in Generation 3 clusters<sup>4</sup> more than 65% of tickets are due to degradable components. Recall from Section 2 that degradable components do not represent a single point of failure as their component count per server is larger than one.

<sup>4</sup>Higher server generations reflect newer server and component architectures. Generation 3 is a currently highly utilized hardware generation.





**Figure 5:** Breakdown of repair tickets at Azure into the component type responsible for the ticket. The repair tickets were recorded on dozens of production clusters spanning regions across two continents and clusters from five different hardware generations, which have been deployed between 2018 to 2022. Most repair tickets in server generations later than 2 are for degradable components and diagnostics indicates a specific pathway.

For degradable component types, Figure 5 further marks in a darker shade the share of tickets that also identify a specific pathway, rather than just the component type. For example, for DIMMs, these tickets would include the specific DIMM slot (recall Figure 3a). We observe that for generations above 2, almost all repair tickets among degradable components also indicate the specific pathway.

The key idea behind FIP is to avoid repair tickets by deactivating (rather than repairing) a faulty degradable component and allowing the server to continue to host customer VMs, albeit with reduced capacity. We refer to this new server state as *degraded* servers.

While Figure 5 illustrates FIP’s potential to reduce repair tickets, a real FIP implementation must also satisfy the following constraints.

- $C_{\text{Performance}}$  VMs placed on degraded servers must still be able to achieve the same peak performance (e.g. memory bandwidth) expected for this VM type (§2).
- $C_{\text{Efficiency}}$  A FIP system must be able to effectively use the capacity on degraded servers. For example, it must not strand one resource (e.g., CPUs) because another resource is degraded (e.g., memory).
- $C_{\text{Capacity}}$  A FIP system must continue to be able to satisfy a region’s demand for VM resources. In particular, VMs must not be turned away from a region because of server degradation or disrepair.

For cloud platforms, FIP system design can be guided by the following observations based on real-world cloud workloads and failure patterns.

First, a majority of VMs that customers are running belong to smaller VM types that can be accommodated on a degraded

| Requested Cores | Core-hours v3 | Core-hours pre-v3 |
|-----------------|---------------|-------------------|
| $\leq 2$        | 27.7%         | 26.9%             |
| (2,4]           | 26.8%         | 16.9%             |
| (4,8]           | 21.5%         | 18.9%             |
| (8,16]          | 10.6%         | 16.6%             |
| $>16$           | 13.4%         | 20.7%             |

**Table 1:** Core counts for VMs introduced with 3rd-generation servers (v3) and with previous-generation servers (pre-v3).

mode server without impacting their performance. For example, Table 1 shows a breakdown of core hours by VM type at Azure. VMs with four or fewer cores account for 40-50% of all core hours and are small enough that they require only a small fraction of a server’s full capacity to achieve their expected performance.

Second, our study of server repair tickets at Azure reveals that the number of component failures per server is typically small compared to a server’s total component count. For example, for servers in Generation 3, 90% of servers that develop SSD and/or DIMM failures in a one-year period exhibit two or fewer failures. The most common failure patterns among those servers are one failed DIMM (36.5%) followed by one failed SSD (10.3%). Hence for the bulk of servers with failures, deactivating the affected components would reduce the server’s capacity by only a small fraction (recall that typical server configurations include 24-32 DIMMs and 6 SSDs) and not cause a significant amount of resource fragmentation. We note however that over long time periods, more than a few components will fail. To prevent resource stranding, any FIP system must thus control how many components can be deactivated in any degraded server.

Third, we find that FIP systems will still have to accommodate some repairs (albeit at a greatly reduced frequency) in order to satisfy capacity requirements. While servers with failures of degradable components are returned to online status, the capacity loss due to servers with failures of undegradable components (which will stay offline in the absence of repairs) is not acceptable.

The design and implementation of a complete FIP system pose multiple open challenges not captured in the simple vision above. For example, FIP requires accurate diagnostics, mechanisms to deactivate component pathways, a detailed understanding of how component deactivation impacts performance, policies to determine when to degrade (versus repair) a server, and a control plane that supports FIP (including the VM scheduler and automated diagnostics).

## 4 Hyrax System Design

Hyrax is a concrete implementation of the FIP idea and the first FIP system at a cloud provider. Hyrax implements a new “degraded” online server state on servers and in the control plane and changes multiple aspects of the offline workflow at Azure. Currently, Hyrax supports three degradable component

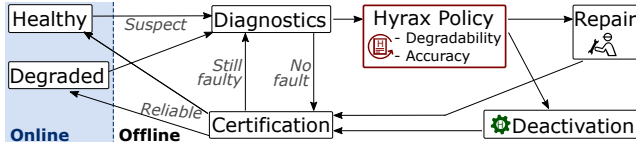


Figure 6: Server states in Hyrax.

types: memory, SSDs, and fans.

Figure 6 provides an overview of server states in Hyrax. After a server is marked as suspect, results from Diagnostics are used by the Hyrax Policy (Ⓜ) to decide whether to degrade or repair. This policy applies first filters for degradable component types. Second, it verifies that diagnostics points to a specific pathway within this component type. Third, it applies a threshold on how many components of each type can be degraded. Degraded servers are created by deactivating the faulty component pathway (⚙️, §5.1). Repairs are scheduled for undegradable component types, when diagnostics cannot identify the faulty component pathway, or if deactivation would cross the policy’s threshold. Degraded and repaired servers are subject to extensive testing (called Certification in §2 and Figure 2) before becoming available for hosting VMs (online).

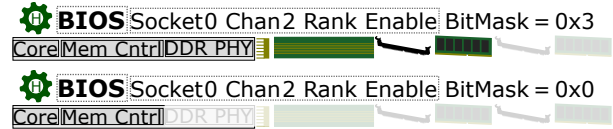
Hyrax achieves  $C_{\text{Efficiency}}$  via the policy’s thresholds. Currently, we never deactivate more than two components of any type. Empirically, we find that this is sufficient to prevent resource stranding. We provide a detailed sensitivity analysis in Section 7.4.

Hyrax achieves  $C_{\text{Performance}}$  by characterizing how deactivating components affects VM performance for different VM types. This allows Hyrax to decide whether the remaining healthy components are sufficient for the server to continue serving VMs and which VM types it can serve without impacting user experience. Hyrax modifies the VM scheduler such that *only the VM types whose performance requirements can be met* are scheduled on the degraded server.

Hyrax minimizes repair tickets because many servers that are degraded instead of repaired will not encounter another fault during their deployed period. If degraded servers encounter another fault that cannot be degraded, Hyrax issues a single repair ticket and technicians repair all faults on the server at once. We call this technique “*mini-batching*”. Mini-batching effectively amortizes technician work like the journey to the server’s rack, identifying and opening the server, manual diagnosis, and record keeping.

Hyrax achieves  $C_{\text{Capacity}}$  in two ways. First, the capacity an individual degraded server can lose is limited via the policy’s thresholds. Second, undegradable servers are not permanently left offline without repairs. We consider a range of different repair schedules (§7).

We discuss technical details of the Hyrax server design in Section 5 and the Hyrax policy and control plane in Section 6.



(a) Deactivating a failed memory component pathway.



(b) Deactivated SSD component pathway.

Figure 7: Component deactivation takes care of entire paths of error sources, such as memory controller, DDR phy, PCB, connector, and DIMM itself. This has the potential to improve over repairs where reseating or exchanging the DIMM often does not resolve the problem.

## 5 Hyrax Servers

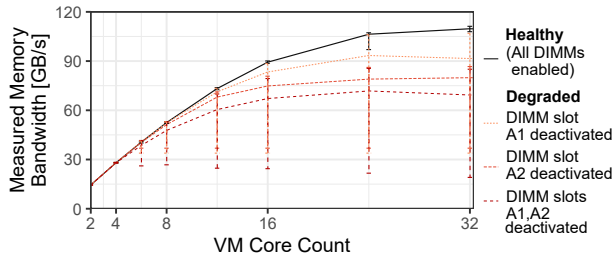
Hyrax seeks to convert an offline server with one or multiple faulty component pathways into a degraded server that can host VMs. Hyrax focuses on memory, SSD, and fans as the most common degradable components (§3). This section describes how to deactivate these three component pathways and associated performance implications.

### 5.1 Component Pathway Deactivation (⚙️)

The key challenge is making component deactivation comprehensive enough so that faults are effectively hidden. Hyrax achieves this by deactivating components using combined firmware and software mechanisms.

**Memory pathway.** Hyrax targets memory errors that cannot be resolved by existing, fine-grained mitigations [8, 13, 35, 55, 67]. Common causes are uncorrectable errors across a DIMM’s banks/ranks, connector problems, or too many faulty rows. Hyrax exploits a rarely-documented firmware (BIOS) feature, called Rank Enable BitMask. On Azure servers, this setting offers a bitmask for each channel, on each memory controller (MC), and on each socket. Each bitmask controls which of the DIMM’s ranks on this channel are included in memory interleaving. Azure diagnostics currently only provides DIMM-level information, so Hyrax deactivation always excludes all ranks on a DIMM. Excluding an entire DIMM means that this DIMM’s memory is not assigned an address. Furthermore, the MC will not attempt to control or refresh any data on that DIMM’s memory chips. Figure 7a shows examples of deactivating one DIMM (0x3) as well as the whole memory pathway (0x0).

Hyrax has two ways to set the Rank Enable BitMask. If the server is able to boot a minimal OS, Hyrax software directly sets the bitmask in the BIOS configuration flash. If the server does not boot, Hyrax can set the bitmask via the Baseboard Management Controller (BMC) on the management network.



**Figure 8:** Peak memory bandwidth of a naïve implementation of Hyrax as measured from VMs on servers with all DIMMs enabled, one DIMM or two DIMMs deactivated, respectively.

**SSD pathway.** Server-local storage for VMs is striped across six NVMe drives. This configuration improves peak performance for IO-intensive VM types and facilitates bin packing hundreds of VM types with server-local storage. We modify the striping software module to read a list of serial numbers to include into the stripe. To deactivate an SSD pathway, Hyrax deletes the drive’s serial number from the striping configuration file. Additionally, we deactivate the SSD component pathways in the BIOS using an option called PCIe Port Config (Figure 7b).

**Fan pathway.** No explicit deactivation is needed for fans. They are monitored by the BMC which emits frequent error messages in case of faults (e.g., zero or low RPM). Hyrax changes BMC firmware to filter out fan error messages for deactivated fan slots.

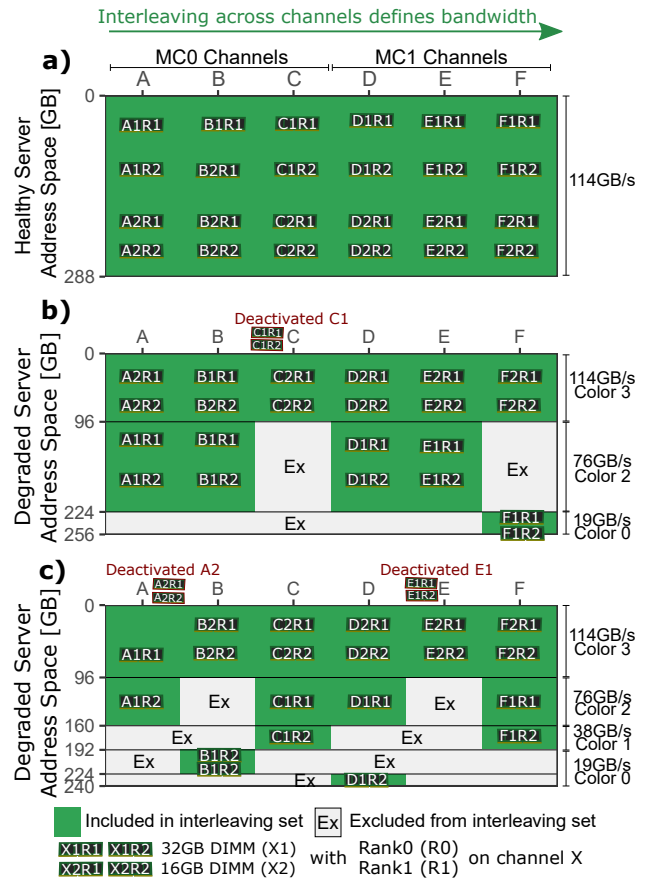
## 5.2 Achieving High Performance on FIP Servers

We describe performance challenges when deactivating memory and SSD pathways and how Hyrax overcomes them.

**Memory pathway.** Cloud servers maximize achievable memory bandwidth by interleaving cachelines across DIMM ranks on all memory channels on the same socket. Deactivating a DIMM limits the processor’s interleaving options and can significantly reduce VM memory bandwidth. Unfortunately, the resulting configuration is almost always outside CPU specifications, known as DIMM population rules [11, 31, 38].

To understand the performance impact of undocumented interleaving from deactivating DIMMs, we experiment with a common production server configuration. This server has two memory controllers per socket (MC0 and MC1), three memory channels per controller (A-C on MC0 and D-F on MC1), and two DIMMs per channel (e.g., A1, A2).

Figure 8 shows memory bandwidth for this server configuration measured in four scenarios: all DIMMs enabled, only DIMM A1 deactivated, only DIMM A2 deactivated, or DIMM A1 and A2 deactivated. We measure the memory bandwidth with a Memory Latency Checker (MLC) [30] for VMs ranging from 4-32 cores and show averages across 10 runs for each VM size. Error bars indicate the worst-performing run



**Figure 9:** Channel interleaving with deactivated DIMMs. The top image shows interleaving for a healthy server, the middle image with DIMM C1 deactivated and the bottom image with two DIMMs (A2, E1) deactivated. Under degraded mode different regions of the address space experience different memory bandwidths, ranging from 19GB/s to 114GB/s.

for each VM size. We observe mean bandwidth loss between 0 to 36% depending on which and how many DIMMs are deactivated. Additionally, we observe that even for the same configuration, there is a significant variance between runs with worst case bandwidth loss up to 82%. Such outliers are not acceptable for deployment. We next explain the underlying reasons and then explain our mitigation.

We find that the inflexibility inherent in channel interleaving is the reason for the bandwidth loss. While a server can have multiple interleaving configurations for different ranks (called sets), each set must either alternate between MCs or just focus on a single MC. Consequently, cross-MC-interleaving requires the same capacity in participating channels on both MCs. To better understand the subtleties involved in interleaving we use a custom firmware debug mode that prints interleaving sets and participating channels. Figure 9 compares the interleaving we observe on healthy versus degraded servers for a single CPU socket on a common platform.

Figure 9a shows interleaving for a healthy server, which

contains a 32GB and a 16GB DIMM per channel<sup>5</sup>. For example, channel A on MC0 contains the 32GB DIMM A1 with ranks A1R1 and A1R2 and a 16GB DIMM A2. Cachelines are interleaved across all six 32GB DIMMs and across all six 16GB DIMMs. Interleaving across all channels creates a uniform address space with 114GB/s, i.e., a sixfold increase over a single channel (19GB/s).

Figure 9b shows a degraded server with C1 (32GB) deactivated. Since symmetry is required within an interleaving set, both C1 and F1 are removed from the first set and as a result, the server interleaves only across the four remaining 32 GB DIMMS. On the other hand, since all 6 16GB DIMMs are still active, the processor continues to interleave across all 6 DIMMs achieving the full 114 GB/s for their part of the address space (note that the 16GB DIMMs now make up the top part of the address space). As F1 is active, but not part of any set so far its capacity remains as non-interleaved (19GB/s). This creates a non-uniform address space with 38% of pages at 114GB/s, 50% at 76GB/s, and 12% at 19GB/s.

A degraded server with two deactivated DIMMs further complicates interleaving sets. Figure 9c shows the interleaving that results when A2 (16GB) and E1 (32GB) are deactivated. With deactivated DIMMs having different sizes, the resulting interleaving sets do not align with full DIMMs and instead use individual ranks (1/2 of a DIMM). The first interleaving set uses A1's first rank (A1R1) and five 16GB DIMMs (B2-F2) achieving the full 114GB/s. The second set uses A1's second rank (A1R2) and the first ranks from DIMMs C1, D1, F1 achieving 76GB/s. The third set uses the second rank from C1 and F1. Two final sets interleave across only a single channel using both ranks from B1 and D1's second rank. This results in an address space with 40% of pages at 114GB/s, 27% at 76GB/s, 13% at 38GB/s, and 20% at 19GB/s.

The main problem with varying peak bandwidth in different address ranges is that it makes VM memory performance on these servers unpredictable. As the OS and hypervisor are unaware of bandwidth differences across the address space the performance of a VM will vary depending on where in the address space its memory gets allocated. A naïve implementation of Hyrax would allocate VMs with a mix of pages leading to low-bandwidth outliers as shown in Figure 8.

To mitigate bandwidth variance on a degraded server, we must know the exact address map that maps address ranges to their achievable peak bandwidth. Unfortunately, reading the interleaving configuration usually requires debugging output that is typically not available. While we can test the memory bandwidth of the entire address space, we found this to be slow and inaccurate. Instead, we conceptually group different deactivation scenarios into equivalence classes, where scenar-

ios in the same class result in the same address map, and store the resulting address map in a distributed database (§6). For example, deactivating a single DIMM leads to two equivalence classes depending on the DIMM size of the deactivated DIMM: The first class includes all scenarios where any one of the 32GB DIMMS fails (and the resulting map would be the image in Figure 9b) and the second class includes all scenarios where one of the 16GB DIMMs fails. Deactivating two DIMMs leads to ten equivalence classes, in addition to two DIMM sizes, interleaving changes with the two DIMMs being on the same channel, within the same MC, in a symmetric or asymmetric position on another MC.

Note that our discussion above focused on only one socket. Since interleaving on different (cache-coherent) CPU sockets happens independently, it is sufficient to characterize one socket. We validated equivalence classes by testing almost all 276 possible combinations. Deactivating three DIMMs leads to 2024 combinations and a multitude of equivalence classes — Hyrax thus deactivates at most two DIMMs and repairs three or more DIMM failures. A sensitivity analysis in § 7.4 will show that disabling larger numbers of DIMMs does also not provide significant gains in terms of repair savings.

Once we know the address map, we employ *page coloring* in the OS/hypervisor memory manager (MM) to assign the same color to pages that are in address regions with equal bandwidth. For example, in Figure 9 we assign colors 0, 1, 2, 3 to pages within a 19, 38, 76, 114 GB/s region, respectively. Each VM type comes with a preferred page color, which is set based on core count. Figure 8 shows that color 0 is sufficient for 2-core VMs. Color 1 is sufficient for 4-6 cores, color 2 for 8-12 cores, and color 3 for above 16 cores. Older generations of VMs sometimes run on new servers, while originally being created for servers with a lower per-channel bandwidth and four (instead of six) channels. Thus, old VM types do not even require color 3 and often use colors 0 and 1.

One could use this coloring scheme to guarantee performance at all times by exposing the amount of available memory for each color to the scheduler. However, to reduce coupling between control plane services, we do not expose this level of detail to the VM scheduler. So, large VMs may be allocated using colors below their bandwidth expectation if no higher colors are currently available on the server. Thus, Hyrax offers only a best-effort guarantee. Empirically, we find that this is sufficient since these cases are exceedingly rare (§7).

**SSD pathway.** The SSD pathway is simple compared to memory. In a fully healthy server, local VM storage is striped across six NVMe drives. VM types are capacity and rate limited (IOPS and bandwidth). When deactivating one NVMe drive, aggregate throughput remains sufficient for even the largest VM type. Deactivating two NVMe drives leads to sufficient throughput for all except the largest VM type. Hyrax thus never schedules this VM type on degraded servers with only four active NVMe drives. Hyrax never deactivates more

<sup>5</sup>The combination of 32GB and 16GB DIMM within one channel is a common configuration to reach target memory-to-core ratios in cloud compute servers of recent years. We discuss this configuration since our experiments with custom firmware happened to run on it. Interleaving on a 32GB/32GB server behaves similarly.



than two NVMe drives and this failure case is rare.

**Fan pathway.** Due to cooling overprovisioning, deactivating up to two fans leads to no performance loss.

## 6 Hyrax Control Plane

The Hyrax control plane consists of two new distributed services that implement the Hyrax policy and many changes to existing control plane services, including the VM scheduler.

### 6.1 Hyrax Policy (🔧)

The Hyrax Policy has two roles (recall Figure 6). First, it interprets diagnostics and sets constraints on which components are degradable. Second, it ensures that degraded servers meet  $C_{Capacity}$ .

To perform the first role, the Hyrax Policy specifies for each server type how many component pathways of each type can be deactivated at once. While Hyrax can adapt to a wide range of thresholds, for our purposes we use two DIMMs, two SSDs, and two fans. These thresholds are guided by common failure scenarios (Section 3) and performance observations (Section 5). Hyrax schedules repairs for any server with more than two faulty component pathways of the same type or any other failure diagnosis. The Policy also includes an extensive mapping list of diagnostic results to valid component pathways. For example, SSD pathways can appear as IO errors, timeouts, and PCIe errors. Diagnostics for SSD failures can sometimes point to PCIe ports and slots that have different (non-SSD) devices or even no device — for these the Hyrax Policy would just schedule the server for repair.

To ensure  $C_{Performance}$ , the Hyrax Policy maps every degraded server configuration to a capacity and performance profile (CPP). The CPP defines the exact server capacity and performance equivalence class (§5). Based on the CPP, Hyrax defines the set of allowable VM types that can run on a degraded server and still meet their SLAs. For example, servers with two DIMMs deactivated on the same channel do not have any page of color 3. This server thus cannot host latest-generation VMs with more than 16 cores. A server with two deactivated SSDs cannot host the largest VM type.

### 6.2 Control Plane

Deactivating component pathways leads to heterogeneous server configurations within a cluster. This requires changes across service and team boundaries. Figure 10 shows a simplified view of Azure’s control plane. We change three and add two new control plane systems.

Let’s consider a server that starts in healthy state and encounters an SSD failure. (1) The Server Health Monitor (SHM) detects NVMe read errors and follows the offlining workflow (§2). (2) Diagnostics reports the SSD component

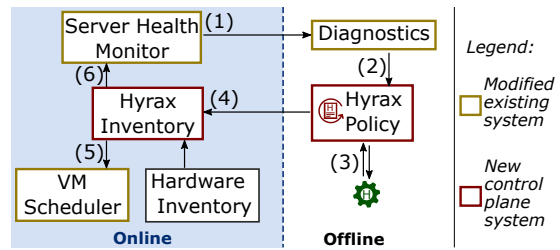


Figure 10: Simplified overview of Hyrax’s control plane.

pathway to the Hyrax Policy (🔧). (3) The policy decides to start the deactivation workflow and communicates with a server-local daemon to deactivate that SSD (⚙️). The deactivated SSD’s serial number is also passed to the new Hyrax Inventory system. (4) After deactivation, the server is tested in the certification step. In the rare event that diagnostics leads Hyrax to deactivate the wrong pathway (§3), it would be detected in this step, e.g., during load testing. After passing certification, the server is onlined. As multiple control plane services might cache the server’s capacity, onlining requires Hyrax to invalidate caches throughout the control plane including the VM scheduler. (5) The Hyrax Inventory shares the server’s capacity and performance profile (CPP) with the VM scheduler (§6.3). Internally, our inventory tracks server state as a delta to the existing Datacenter Inventory. The delta consists of the serial numbers and slots of deactivated components, which remains small enough to fit into a single inventory server’s memory. (6) The Hyrax Inventory sends active serial numbers and slots to the SHM. The SHM only checks for these active components, which prevents the SHM from triggering warnings over missing components which have been deactivated.

There are additional changes in downstream services not shown here. For example, it was previously uncommon for servers to have multiple concurrent failures, so repair tickets used to be issued only for a single component type. With Hyrax, it is common for repair tickets to include multiple different component types. For example, there are no tickets for a server with two DIMM failures. However, if the two DIMM failures are later followed by any failures for an undegradable component (e.g., the NIC) the repair ticket will involve two different component types. To minimize repair tickets, Hyrax changed the ticket workflow and retrained technicians to repair multiple different component types at once, with a single ticket (mini-batching).

### 6.3 VM scheduling policy

Hyrax requires three changes to VM scheduling and an optional optimization. First, the VM scheduler consumes Hyrax Inventory to calculate hardware resources for individual servers instead of a single lookup to obtain a cluster’s homogeneous server type. The overhead of this lookup is negligible as servers moving from offline to online state is

| All Tickets (100%)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                       |                            |                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|----------------------------|---------------------------|
| Diag says undegradable (27%)                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                       | Diag says degradable (73%) |                           |
| Diag in-accurate (2.7%)                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Accurate Diag (24.3%) | Diag in-accurate (0.9%)    | Accurate Diag (72.1%)     |
| <b>Legend</b><br><span style="display: inline-block; width: 10px; height: 10px; background-color: #ffffcc; border: 1px solid black; margin-right: 5px;"></span> Possibly unnecessary server repair ticket (missed opportunity)<br><span style="display: inline-block; width: 10px; height: 10px; background-color: #ffcccc; border: 1px solid black; margin-right: 5px;"></span> Server incorrectly restarted in degraded mode (negative user experience, if not caught by certification testing) |                       | Missing location (2.8%)    | Diag has location (69.3%) |

**Figure 11:** Accuracy of automated fault diagnostics at Azure and their impact on Hyrax.

rare compared to VM scheduling events.

Second, we extend filter rules (§2) to enforce Hyrax’s CPP, i.e., which VM types can be placed on every server.

Third, we change the definition of a cluster’s “*capacity reserve*”. The capacity reserve exists for multiple reasons, including to have a target to migrate VMs to when a server shows signs of failing soon. A key component of the capacity reserve is to have some healthy empty servers (HES) that are able to host any kind of VM, including full-server VMs that use all of a server’s capacity. Degraded servers are not able to host all full-server VMs. We thus exclude them from being counted as HES.

Finally, we change a preference rule to optimize scheduling. Since degraded servers cannot be counted as HES, we prefer fully-healthy servers to become empty and stay empty. Our change updates rules to prefer placing VMs on degraded servers over healthy servers, provided no other rule takes precedence. By doing this we increase HES counts which allows placing more VMs into clusters.

## 6.4 Hyrax Diagnostics

Hyrax builds on an existing automated monitoring and diagnostics system. This system’s output is targeted at humans and includes information on which component type is faulty and its location. To use this system, we add an interpreter that maps diagnostic results to valid Hyrax component pathways. As part of this design, we analyzed four years of repair ticket logs at Azure. This analysis shows the *accuracy* of the diagnostic system and how Hyrax handles inaccurate or incomplete diagnoses. Specifically, we rely on notes from human technicians, who worked on the tickets in our history of repair logs. These notes indicate whether the diagnosis was correct, including whether the right component was identified.

At a high level, we find that diagnostic accuracy is high. For example, across all tickets in 2021, 96.4% accurately identify the component type at fault. For a more detailed view, Figure 11 shows a breakdown of all diagnoses made in 2021, outlining the different scenarios that arise and how they

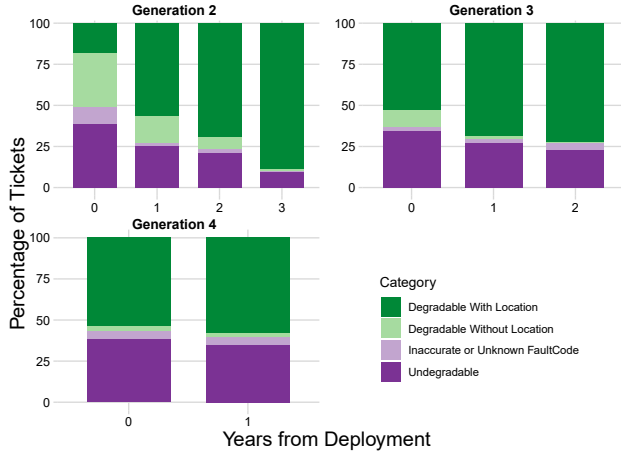
impact Hyrax’s operation.

We make two interesting observations: First, diagnostic accuracy is lower for diagnoses pointing to an *undegradable* component: 10% of tickets labelled with an undegradable component are inaccurate (accounting for 2.7% of *all* tickets). Fortunately, this type of misdiagnosis is relatively benign. Hyrax will take the server offline (for potential later repair), which is the intended behavior if the actual faulty component is indeed undegradable. It is however a *missed opportunity* to keep the server running in degraded mode if the true fault is in a degradable component.

Second, diagnostic accuracy is very high for diagnoses pointing to a degradable component: 98.8% of tickets labelled as degradable do accurately identify the component type at fault. Within these, some diagnoses are *incomplete*, where the correct component type is specified, but location information is missing (e.g. the diagnosis indicates a DIMM problem, but does not specify a DIMM slot). More precisely, 3.8% of the accurately diagnosed degradable tickets (corresponding to 2.8% of *all* tickets) are missing location information which leads Hyrax to offline the server despite the fact that the faulty component is degradable. These tickets thus also represent a *missed opportunity* for degraded mode operation.

The last scenario we need to consider is the 1.2% of degradable tickets that contain an inaccurate diagnosis pointing to the wrong component type. These make up only 0.9% of all tickets, but their impact on Hyrax is less obvious. In the best case, Hyrax will try to deactivate the specified pathway and certification testing (recall §4) fails since this is not the faulty component. Failing certification testing with any degraded component automatically triggers an investigation both by a technician and by the Hyrax on-call team. In the worst case, the server passes certification testing and returns to serve customer VMs despite the fact that the true faulty component has not been degraded or repaired. This can lead to negative user experience as VMs may be scheduled on the server and they may get interrupted if the server is offlined again. Such repeat offlining of the same server also happens for technician repairs. In fact, our preliminary data indicates that the rate at which repaired servers are offlined again is comparable to such inaccurate decisions by Hyrax. This is likely because technicians rely on the same automated diagnostics and certification process as Hyrax.

We conclude by noting that diagnostic accuracy has continuously improved over the past years. Figure 12 shows the breakdown of repair tickets for three different hardware generations (Gen 2-4) by year since deployment. We observe that accuracy has improved from generation to generation, and also that accuracy improves over time within a particular hardware generation. Both the fraction of tickets with missing location and tickets with inaccurate fault code have decreased over the years. The reason is a concerted effort by the diagnostic team at Azure to add more coverage of various fault codes as well as improvements based on technician feedback.



**Figure 12:** The progression of the breakdown of repair tickets at Azure by deployment year for three generations of servers.

## 7 Evaluation

### 7.1 Evaluation Setup

We use two types of setups in our evaluation of Hyrax. First, we evaluate Hyrax on production servers to characterize its performance and ability to mitigate faulty components. Our evaluation focuses on 3rd-generation servers which have been deployed for 2-3 years. Second, to measure cluster-level impacts on repairs and VM scheduling over six years, we use trace-driven large-scale cluster-level simulations.

#### 7.1.1 Server experiments

We use production server hardware and *synthetically inject failures* using a commercial memory error injector (MEI) that interposes on the DDR memory bus [29]. We also perform *real failure* tests by intercepting nodes after Diagnostics flags a memory fault, but before a repair ticket is issued (§4).

We measure latency and bandwidth with Intel MLC [30] from inside VMs on healthy and degraded servers. MLC characterizes worst-case performance as it is more sensitive to deteriorated latency and bandwidth than any real-world application we’ve tested. We compare three implementations.

- **Hyrax:** Coloring approach based on 1GB hypervisor page table entries (§4)
- **Naïve:** Hypervisor randomly allocates VM memory among free pages
- **Interleaving:** 4kB-interleaving in hypervisor page tables

Our tests cover Intel servers from generations 3-5 and a subsequent (not yet deployed) generation. We report measurements from the 3rd generation as results from other generations are qualitatively the same. A typical 3rd-generation server uses two Intel Skylake processors (96 threads total). Each socket is equipped with six DDR4 channels with a 32GB and a 16GB DIMM per channel. Memory interleaving is enabled across all ranks on the same socket; thus, the

OS/hypervisor sees two NUMA nodes. There are six data SSDs using 960GB NVMe drives. The server runs Azure’s production-grade hypervisor and software stack. VMs are allocated with a 1GB page size.

#### 7.1.2 Large-scale simulations

We replay VM, failure and repair ticket traces in a simulated environment, using the Azure production VM scheduler code base. The traces span 66 clusters that host general-purpose VMs from regions in the US and Europe. With only 2-3 years of real failure traces for 3rd-generation servers, we model future failures with the help of 1st and 2nd-generation failure traces. The simulator models Hyrax’s control plane components (Figure 10) including Hyrax and all server states (Figure 6).

We compare two designs.

- **Hyrax:** Hyrax enables degraded server states and repairs servers with undegradable components and above thresholds (§4).
- **AoN:** All-or-Nothing repairs all hardware faults.

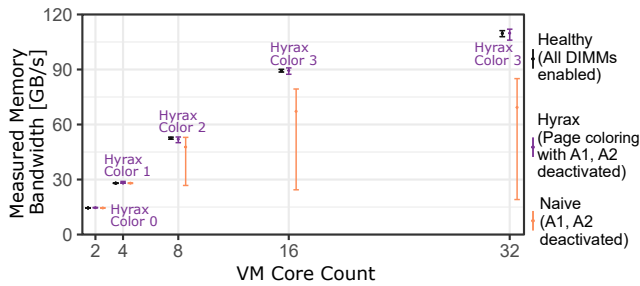
We simulate four possible repair schedules: issuing an immediate repair ticket (**IR**) and scheduling batch repairs every 3, 6, or 12 months (**3m**, **6m**, **12m**). For IR, we sample actual repair delays from Azure production datacenters. For batch repairs, we assume a hypothetical schedule where repairs are immediately effective at 3, 6, or 12 months. This batch repair schedule is unlikely how batch repairs would actually be implemented in practice. Instead, its purpose is to show a hypothetical and simplified schedule that could also reduce repair work, to highlight the impact of degraded mode operation. For each repair schedule, we compare the number of repair tickets, repair trips, resource availability and impact on arriving VMs under Hyrax and AoN.

We cross-validate the simulator for AoN relative to real-world clusters with the same failures and VM workloads. Due to the inherent randomness in placement decisions, repeated runs have small deviations. Across runs on 10 clusters, simulation of AoN and real-world metrics are within 0.25%. Overall, our simulations required more than 80,000 CPU hours.

### 7.2 Correctness

In this section, we use production server measurements to demonstrate that Hyrax can correctly deactivate component paths and thereby avoid future faults on a path. Due to space constraints, we focus on memory faults and omit qualitatively-similar SSD experiments.

**Synthetic failures.** We measure memory error rates with the MEI placed on a given DIMM slot and either activate all ranks (no-Hyrax) or deactivate the corresponding slot (Hyrax). We target the MEI to corrupt bits matching a single row address and start a VM on the same CPU socket. The VM runs MLC in the peak bandwidth setting. Under no-Hyrax, we observe



**Figure 13:** Peak memory bandwidth of a healthy server, a Hyrax server with page coloring, and a naïve implementation of degraded servers with two DIMMs deactivated.

a high rate of correctable memory errors. There are bursts of uncorrectable errors that lead to both VM and host crashing within minutes. With Hyrax, there are no memory errors throughout the duration of a 48 hour test; the VMs and the host run without errors or crashes.

**Real-world failures.** We identify a server in a test cluster that was diagnosed with a high rate of uncorrectable memory errors on one DIMM. Diagnostics is able to boot its minimal OS and reproduce these memory errors. Hyrax recognizes that this server can be degraded and deactivates the correct DIMM. Certification testing does not find any memory errors and issues a “pass” that qualifies this server for hosting VMs.

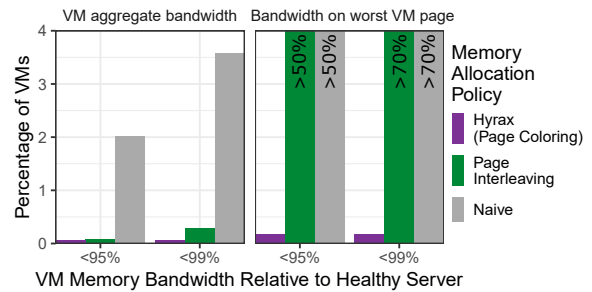
### 7.3 Performance

In this section, we demonstrate that Hyrax can successfully mitigate any VM performance impact of degraded mode operation. For space reasons, we focus on the more complex case of memory performance (memory latency and bandwidth).

**Server-level experiments.** Figure 13 compares VM memory bandwidth of Hyrax and Naïve on a degraded server to a healthy server. The degraded server has A1 and A2 deactivated. Hyrax allocates the VM using colors 0-3, depending on VM core count (§4). We find that memory bandwidth under Hyrax is within 1% of the healthy server. In contrast, Naïve’s performance is highly variable with mean bandwidth up to 36% lower and worst-case bandwidth up to 82% lower than on the healthy server.

We also tested memory latency. In all three systems, and across all experiments, the unloaded memory latency reported by MLC for the degraded server remains within 5% of the healthy server.

**Large-scale page coloring simulations.** The previous experiment focused on a single VM in isolation for one particular failure pattern. For a more complete view of VM performance under Hyrax we use simulations that are driven by actual traces of VM arrivals and departures to capture the effect of VM churn and also simulate component deactivation based on real failure traces to capture the rich set of failure patterns



**Figure 14:** Hyrax almost always achieves the same VM memory bandwidth on degraded nodes as VMs would on healthy servers.

that arises in practice. (The traces come from our cluster simulations in §7.4). We play back these VM events in server-level simulations of the three memory allocation policies: Hyrax page coloring, page interleaving, and Naïve.

Figure 14 shows the percentage of VMs with less than 95% and 99% of the bandwidth of a healthy server, both for VM aggregate bandwidth (left) and bandwidth of the VM’s *worst* page (right). With Hyrax, fewer than 0.16% of VMs see bandwidth on their worst page that is lower than 99% of the worst-page bandwidth achieved on a healthy server. VM *aggregate* bandwidth under Hyrax is even closer to that of a healthy server.

Page interleaving also results in a low percentage of VMs that achieve less than 95-99% of the aggregate memory bandwidth of a healthy server. However, more than half of VMs include at least one memory page with significantly lower bandwidth. We also note that page interleaving increases a VM’s page table by orders of magnitude. This leads to a high rate of TLB misses and increased memory access latency. In practice, we know that memory access latency is even more important than bandwidth — internal production workloads lose 5-15% of performance for small page sizes. Thus, interleaving is not practical.

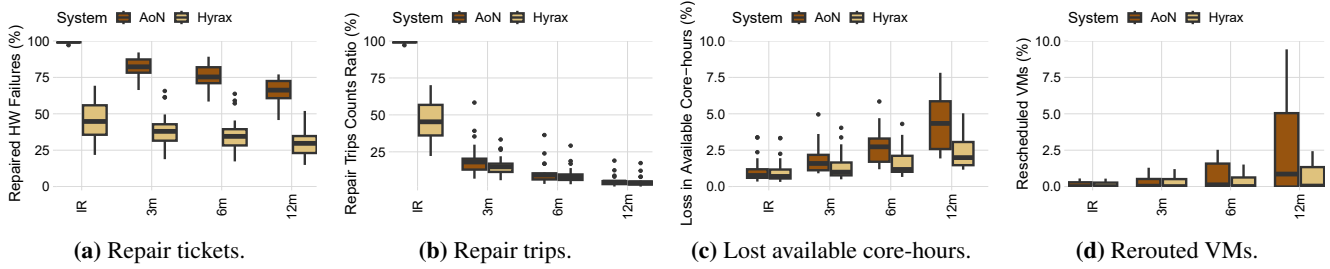
Naïve is compatible with large page sizes but more than 2% of VMs achieve less than 95% of the aggregate bandwidth goal. This grows to 3.5% for a goal of 99% and above 50% when considering the worst page in a VM. While Naïve performs well on average, tail performance matters at scale.

### 7.4 Large-scale Cluster Simulations

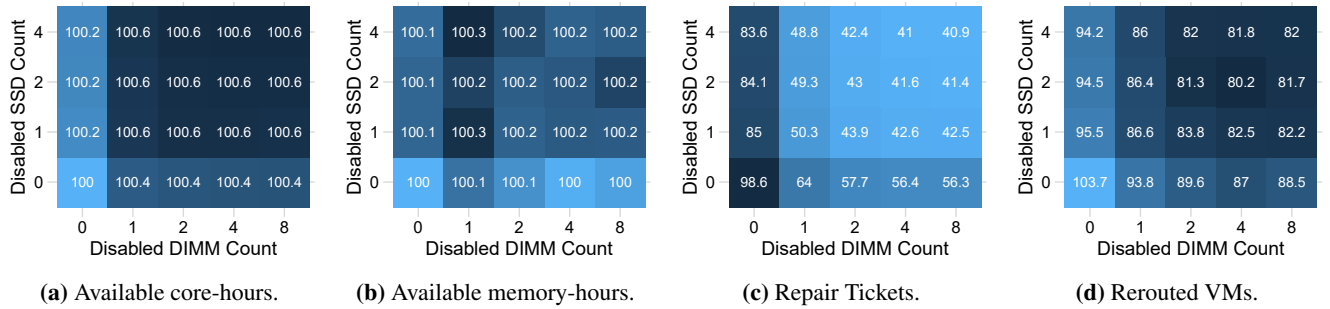
We turn to large-scale cluster simulations to characterize Hyrax’s impact on repair tickets, repair trips, cluster resource availability and user impact. We consider four different repair modes: Azure’s process of immediately scheduling a repair ticket (IR) and hypothetical repair batching policies with three different intervals (3 months, 6 months, 12 months).

**Repair tickets.** We begin by measuring for each repair mode the percentage of all hardware failures that result in a repair ticket, i.e. the failures that require a technician to perform





**Figure 15:** Results from a simulated deployment of Hyrax across two regions with 66 compute clusters and four repair schedules: immediate repair tickets (IR) and batch repairs (3m, 6m, 12m). The figures compare key metrics under Hyrax with all-or-nothing server operation (AoN).



**Figure 16:** Ratio between Hyrax and AoN for different threshold settings with fixed batch repair interval (3m). Stranded availability is not included when computing core-hours and memory-hours availability.

physical examinations and repairs. Figure 15a shows the results for Hyrax and AoN using boxplots, where each data point in the distribution represented by the boxplot corresponds to one of the 66 clusters.

We observe that Hyrax reduces the number of repair tickets by more than a factor of 2 across all repair modes. Both mean and median are consistently around 55% lower under Hyrax than under AoN. A significant contributor to Hyrax’s effectiveness is mini-batching. Specifically, under Hyrax, we find that 56% of repair tickets contain more than one component, compared to single-digit fractions for AoN.

**Repair trips.** We compare the number of repair trips required under Hyrax and AoN, i.e. the number of times when a technician needs to travel to a cluster. Figure 15b shows the number of repair trips normalized by the number of hardware failures.

We observe that Hyrax significantly reduces repair trips under immediate repairs (IR). While under AoN every hardware failure results in a repair trip, under Hyrax, on average 55% of these repair trips can be avoided by deactivating the affected component.

Under batch repairs, the number of repair trips to a cluster is upper-bounded to once every  $x$  months, where  $x$  is the repair interval. Interestingly, Hyrax still provides improvements over AoN, albeit smaller than for IR. For example, for a batch repair interval of 3 months Hyrax reduces repair trips by around 20% (mean and median across clusters). Every saved repair trip results from a 3 month interval in which Hyrax was able to

handle all failures with component deactivation.

**Lost available core-hours.** This metric quantifies the impact of the repair operating model on the availability of cluster hardware resources. In particular, we consider the percentage of a cluster’s total core-hours (i.e. number of cores in the cluster multiplied by cluster lifetime) that are *lost*, i.e., a core physically exists in the cluster, but is not available to run VMs due to one of two reasons: (1) A server is offline for repairs; (2) Due to resource fragmentation some of a server’s cores cannot be allocated to VMs because of limited availability of another resource (DIMMs or SSDs) [40]. Hyrax might exacerbate resource fragmentation as it might deactivate multiple components of one type, making it harder to utilize the remaining components.

Figure 15c shows that under a batch repair schedule Hyrax significantly reduces the loss of available core-hours. Hyrax keeps servers running (albeit with reduced capacity) after degradable component failures, rather than taking the entire server offline until the next scheduled batch repair. The improvement in the median lost core-hours of Hyrax over AoN ranges from 38% for a 3m interval to 55% for a 12m interval.

Interestingly, we observe that Hyrax improves loss in available core-hours even in the IR repair schedule. The median loss in core-hours is 9% lower under Hyrax than AoN. The reason is that immediate repairs are not truly immediate - typical repair times are on the order of days, but can sometimes take much longer, depending on component availability. In

contrast, deactivating components is consistently fast.

**Rerouting of VMs.** When a cluster’s available resources are insufficient to host an arriving VM, the VM is rerouted to a different cluster. Rerouting of VMs can negatively impact user experience as it increases the time until a VM gets started. Figure 15d shows the percentage of arriving VMs that are being rerouted under Hyrax versus AoN.

Under IR, the fraction of VMs that get rerouted is very small for both Hyrax and AoN. While it is identical (zero) in the median for both policies, the mean is slightly lower (4% reduction) under Hyrax as it decreases lost core hours in some clusters with many failures.

When moving to batch repair schedules, Hyrax provides clear improvements over AoN, ranging from an average 38% reduction in rerouted VMs for 3 month batch repairs to an average 64% reduction for 12 month batch repairs. These improvements are a direct consequence of the reduced loss in system capacity (core-hours) under Hyrax compared to AoN.

**TCO impacts.** Server repairs account for 9 to 12% of TCO (§2), and Hyrax reduces repair tickets by an average of 55% across the simulated clusters (§7.4). However, repairs frequently involve multiple components as well as undiagnosed failures, which extends repair times by about 15%. Thus, Hyrax reduces technician time by about 48%, which translates to a 4.5 to 6% reduction in TCO.

**Sensitivity to Hyrax’s deactivation thresholds.** Our implementation of Hyrax chooses its deactivation threshold of two per component type to reduce complexity. Figure 16 shows how different choices of thresholds impact available core hours, available memory hours, repair tickets and rerouted VMs. The numbers in the figure represent the ratio of Hyrax to AoN for a batch interval of 3m. Darker color shading corresponds to better results.

We observe that available system capacity (core-hours as well as memory hours) does not improve/change significantly beyond a threshold of one DIMM and one SSD. The reduction in number of repair tickets and rerouted VMs under Hyrax continues to increase as thresholds increase, however, returns are diminishing past a threshold of two DIMMs and two SSDs. One of the reasons is that it is very rare that more than two DIMMs and/or more than two SSDs fail in the same server, so these scenarios have little impact on key metrics.

In conclusion, increasing the thresholds beyond two per component type provides very limited gains while increasing system complexity, e.g. in handling servers with very low performance due to a large number of degraded components.

**Sensitivity to different regions.** Figure 1 shows that Hyrax performs similarly across the US and EU region.

**Sensitivity to server generation.** We also simulated a full deployment of Hyrax on 4th-generation servers. Figure 12 shows that this generation has an overall lower percentage of degradable components. Hyrax’s benefits are thus slightly less pronounced on this server generation. However, as diagnostics

has improved over time for 2nd and 3rd-generation servers, 4th-generation servers may also improve in the future.

## 8 Related Work

Our work is the first work to explore degraded mode operation in the context of VM compute servers and at the scale of a cloud platform.

**Datacenters that fail-in-place.** Related to our work are the general efforts toward lights-out data centers such as containerized datacenters [23, 65], underwater datacenters [10], and zero-maintenance storage systems [49, 50]. In our evaluation, AoN with high batch repair intervals (12m) represents these approaches. Unfortunately, the loss in availability or cost (hardware, power, space) to make up for this loss is prohibitive without degraded mode.

**Mechanisms to implement fail-in-place.** We borrowed the term degraded mode from RAID systems [51], where upon failure of a drive, the system seamlessly continues to operate until the failed drive is replaced, however at *reduced capacity and reduced performance*.

There are many existing fault-tolerance approaches that use component-internal redundancy [8, 25, 34, 44, 52, 55, 67, 68]. Hyrax targets the left-over failures not already covered by these approaches. It can be viewed as taking degraded mode to the extreme and applied to even combinations across different devices. As such, Hyrax has different requirements that raises novel challenges (§4).

**Improving repairs and redundancy.** Recent efforts for reducing the reliance on human technicians in lights-out datacenters explore the use of robots to replace hardware components [56]. Currently, this technology is not sufficiently capable, versatile and economical to be employed at scale. Our work presents a solution that can be deployed immediately in today’s systems.

Finally, systems that require no or minimal repairs throughout their lifetime are common in the context of embedded systems, for example, as part of autonomous vehicles, airplanes or satellites [6, 14, 47, 73]. However, these are special purpose systems with specialized components and significant redundancy. In contrast, we are exploring whether a cluster based on commodity data center components can operate with no or minimal repair throughout its lifetime through the use of fail-in-place.

## 9 Deployment Experience and Discussion

Hyrax reduces repair tickets while maintaining cluster capacity, VM scheduling, and VM performance. We discuss deployment experience and broader issues.

**Deploying incrementally.** Hyrax requires changes across teams that have not previously interfaced, including hypervi-

sor software engineers, hardware validation teams, and data-center staffing. Such a large project requires years to achieve visibility and alignment. During this process, we developed variants of Hyrax that could be deployed incrementally and fly largely under the radar. Our first increment focused on deactivating a single SSD in clusters without impacts on VM scheduling due to spare capacity and bandwidth. Further, we shortcut offline state changes as we could pinpoint some SSD failures without diagnostics. Our shortcut quickly migrated VMs away, rebooted the server, and deactivated the faulty SSD without leaving the online server state. Building in increments increased visibility and buy-in across Azure which facilitated far-reaching changes to VM scheduling and offline server workflows. Overall, Hyrax demonstrates the feasibility of overcoming ossification in large software stacks.

#### **Reduced benefits due to non-optimized software paths.**

The server state diagram in Figure 6 and the control plane overview in Figure 10 are vastly simplified. In principle, a degradable server should be able to return online within half an hour (after a reboot). However, before Hyrax, repairs took multiple days and sometimes even weeks, e.g., due to supply chain issues. Thus, the duration of offline states and transitions did not matter. Under Hyrax, returning to online has to wait for these states, which takes multiple hours in production.

An early variant shortcuts the offline state and returned servers to online within minutes. Unfortunately, the deployment scale of this variant is limited as few faults can be recognized as degradable without deep diagnosis. The limited scale of the shortcut variant and the slowness of Hyrax's offline implementation currently limits Hyrax's ability to improve cluster capacity. This is reflected in our simulations (§7).

**The usefulness of simulations and quantitative data.** We tested significant parts of the production code for inventory and state management in a mocked-up environment driven by simulated failures. Our large-scale simulations also helped convince engineering teams to help with large-scale changes. For example, we initially faced significant skepticism towards mini-batching. This was partly due to multiple past efforts that had tried and failed to implement mini-batching. These past efforts had cemented the idea that multiple components failing at once is a very rare occurrence. Simulations showed that Hyrax led to a high occurrence of mini-batched tickets.

**Tailoring automated diagnostics for FIP.** While our work shows that FIP can work with existing diagnostics systems, there is still room for improvement, including fine-grained diagnostics to find individual faulty cores and to improve locating other component paths (§6.4). We also find subtle shortcomings in diagnostics systems due to their focus on technician repairs. For example, current diagnostics systems prefer not to issue a ticket when they cannot reproduce a failure and pinpoint a specific repair action. This is required due to the high cost of false positives, i.e., calling a technician and replacing a component when the underlying component

was not actually faulty. The flip side is a higher rate of false negatives, which we observe as repeated failures on the same server. Hyrax's automation may open up a path towards improving cloud reliability and availability. Specifically, a FIP system could tolerate a higher rate of false positives (as they lead to a negligible capacity impact), and in exchange achieve lower false negative rates.

**Interaction with class failures.** An early practical concern at Azure was how Hyrax interacts with the occurrence of class failures, which is a recall of a large set of components of similar types from the same manufacturing period. Over three years, we found class failures affecting multiple PSU, DIMM, and CPU models, and one SSD model. Class failures often lead to an expectation of increased failure rates which may affect availability. Thus, associated components are typically proactively swapped out for new components. While class failures cause only about 5% of repair tickets, they often affect a large percentage of servers in the same cluster at once. If the number of affected components in a server is below Hyrax's thresholds, degraded mode can be an effective mitigation. However, deactivating many components at once may negatively affect VM scheduling. Thus, when we look back at three years of class failures, Hyrax would have only been effective in mitigating one out of about a dozen of class failures.

**Implications for new datacenter environments.** Our findings affect how one might design a future datacenter. In short, Hyrax reduces repair needs but does not obviate the need for repairs entirely. Specifically, the capacity loss after 6 to 10 years of deployment without repairs exceeds the cost savings of most new datacenter designs. We thus expect to see continued need for individual component replacement.

Hyrax's reduction in the number of repairs may be sufficient to offset the additional repair time introduced by some designs, such as new cooling techniques [33, 72]. Specifically, we find that Hyrax enables datacenter designs that result in repairs that take about twice as long. When repair times take much longer, TCO will increase even with Hyrax. This might be the case for some server and datacenter designs including extremely dense servers [15–18, 21, 27, 28, 36], connector-less server designs with soldered-on components [45], or datacenters in hard-to-reach locations, such as sealed containers on the ocean floor [10].

## **Acknowledgments**

We thank our shepherd, Daniel Peek, and the anonymous OSDI '23 reviewers for their great comments. We thank our many partner teams within Microsoft including Dirk Hofmann, Saptadeep Chanda, and Tom Harpel for their continued support on understanding technician training, workflows, and staffing; Rama Bhimanadhuni for his help on firmware; and Manish Dalal for early feedback on interpreting diagnostics.

## References

- [1] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020. 2
- [2] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafir, and Assaf Schuster. viommu: Efficient iommu emulation. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. 2
- [3] Backblaze. Hard drive data and stats. <https://www.backblaze.com/b2/hard-drive-test-data.html> accessed 6/26/2022, June 2022. 1
- [4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013. 1, 1, 2, 2
- [5] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003. 1
- [6] John W Bennett, Glynn J Atkinson, Barrie C Mecrow, and David J Atkinson. Fault-tolerant design considerations and control strategies for aerospace drives. *IEEE Transactions on Industrial Electronics*, 59(5):2049–2058, 2011. 8
- [7] Daniel S. Berger, Fiodar Kazhamiaka, Esha Choukse, Íñigo Goiri, Celine Irvine, Pulkit A. Misra, Alok Kumbhare, Rodrigo Fonseca, and Ricardo Bianchini. Research avenues towards net-zero cloud platforms. Workshop on NetZero Carbon Computing, 2 2023. 1, 2, 2
- [8] Stuart Allen Berke and Vadhira Sankaranarayanan. System and method for post-package repair across dram banks and bank groups, August 2019. US Patent 10,395,750. 2, 5.1, 8
- [9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016. 2
- [10] Ben Cutler, Spencer Fowers, Eric Peterson, and Mike Shepperd. Project natick. OpenCompute OCPREG19 track on Rack & Power / Advanced Cooling <https://natick.research.microsoft.com/> accessed 6/26/2022, October 2020. 8, 9
- [11] Dell. Memory population rules for 3rd generation intel xeon scalable processors on poweredge servers. <https://www.delltechnologies.com/asset/en-us/products/servers/industry-market/whitepaper-memory-population-rules-for-3rd-generation-intel-xeon-scalable-processors-on-poweredge-servers.pdf> accessed 11/26/2022, 2022. 5.2
- [12] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621, 2014. 1
- [13] Xiaoming Du and Cong Li. Combining error statistics with failure prediction in memory page offlining. In *International Symposium on Memory Systems*, pages 127–132, 2019. 5.1
- [14] Elena Dubrova. *Fault-tolerant design*. Springer, 2013. 8
- [15] E3NV. Ots immersion servers. <https://www.e3nv.com/immersion-servers> accessed 6/26/2022, 2022. 9
- [16] Wesley M Felter, Tom W Keller, Michael D Kistler, Charles Lefurgy, Karthick Rajamani, Ramakrishnan Rajamony, Freeman L Rawson, Bruce A Smith, and Eric Van Hensbergen. On the performance and use of dense servers. *IBM Journal of Research and Development*, 47(5.6):671–688, 2003. 9
- [17] Gigabyte. Coolit liquid-cooled ready servers. <https://www.gigabyte.com/Industry-Solutions/coolit-liquid-cooled-ready-servers> accessed 6/26/2022, 2022. 9
- [18] GRC. Servers designed for immersion (sdi). <https://www.grcooling.com/servers-for-immersion-cooling/> accessed 6/26/2022, 2022. 9
- [19] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks, 2008. 2
- [20] Albert Greenberg and Dave Maltz. What goes into a data center. SIGMETRICS 2009 Tutorial, 2009. 2
- [21] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G Dreslinski, Luis Ceze, and Trevor Mudge. Integrated 3d-stacked server designs for increasing physical density of key-value stores. In *ACM ASPLOS*, pages 485–498, 2014. 9
- [22] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, David Dion, Esaias E Greeff, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean:vm allocation service at scale. In *USENIX OSDI*, pages 845–861, 2020. 2



- [23] James R. Hamilton. An architecture for modular data centers. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 306–313. [www.cidrdb.org](http://www.cidrdb.org), 2007. 8
- [24] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021. 1, 2
- [25] Duwon Hong, Myungsuk Kim, Geonhee Cho, Dusol Lee, and Jihong Kim. Guardederase: Extending ssd lifetimes by protecting weak wordlines. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 133–146, 2022. 2, 8
- [26] Amy Hood. Microsoft earnings release fy22 q4. <https://www.microsoft.com/en-us/Investor/earnings/FY-2022-Q4/press-release-webcast> accessed 11/26/2022, 2022. 2
- [27] Hypertec. Trident immersion servers. <https://hypertec.com/ciara/immersion-servers/> accessed 6/26/2022, 2022. 9
- [28] AVNET Integrated. Integrated rack with immersed, liquid-cooled it. <https://www.avnet.com/wps/portal/integrated/resources/liquid-cooling/> accessed 6/26/2022, 2022. 9
- [29] Intel. Memory error injection mei test card and utility. [https://designintools.intel.com/MEI\\_Test\\_Card\\_and\\_Utility\\_p/stlgrn61.htm](https://designintools.intel.com/MEI_Test_Card_and_Utility_p/stlgrn61.htm) accessed 6/26/2022, 2017. 7.1.1
- [30] Intel. Memory latency checker v3.9a. <https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html> accessed 6/26/2022, 2022. 5.2, 7.1.1
- [31] Intel. Supported memory and memory population rules for the intel server board family. <https://www.intel.com/content/www/us/en/support/articles/000055509/server-products/server-boards.html> accessed 11/26/2022, 2022. 5.2
- [32] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007. 1, 2, 2
- [33] Majid Jalili, Ioannis Manousakis, Íñigo Goiri, Pulkit A Misra, Ashish Raniwala, Husam Alissa, Bharath Ramakrishnan, Phillip Tuma, Christian Belady, Marcus Fontoura, et al. Cost-efficient overclocking in immersion-cooled datacenters. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 623–636, 2021. 1, 9
- [34] Dae-Hyun Kim and Linda S Milor. Ecc-aspirin: An ecc-assisted post-package repair scheme for aging errors in drams. In *IEEE VLSI Test Symposium*, pages 1–6, 2016. 2, 8
- [35] Andi Kleen. Mcelog bad page offlining. <http://www.mcelog.org/badpageofflining.html>, 2021. 5.1
- [36] Ravi Kollipara, Ming Li, Chuck Yuan, Hideki Kusamitsu, and Toshiyasu Ito. Evaluation of high density liquid crystal polymer based flex interconnect for supporting greater than 1 tb/s of memory bandwidth. In *2008 58th Electronic Components and Technology Conference*, pages 1132–1138, 2008. 9
- [37] Alok Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Vieira Frujeri, Nithish Mahalingam, Pulkit Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-based power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 473–487, 2021. 2
- [38] Lenovo. Balanced memory configurations with second-generation intel xeon scalable processors. <https://lenovopress.lenovo.com/lp1089.pdf> accessed 11/26/2022, 2022. 5.2
- [39] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafir. Page fault support for network controllers. In *ASPLOS*, pages 449–466, 2017. 2
- [40] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023. 7.4
- [41] Fan Lin, Matt Beadon, Harish Dattatraya Dixit, Gautham Vunnam, Amol Desai, and Sriram Sankar. Hardware remediation at scale. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 14–17. IEEE, 2018. 1, 2, 2
- [42] Zitao Liu and Sangyeun Cho. Characterizing machines and workloads on a google cluster. In *International*

*Conference on Parallel Processing Workshops*, pages 397–403, 2012. 1, 2

- [43] Stathis Maneas, Kaveh Mahdavian, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 137–149, Santa Clara, CA, February 2020. USENIX Association. 1
- [44] Stathis Maneas, Kaveh Mahdavian, Tim Emami, and Bianca Schroeder. Reliability of ssds in enterprise storage systems: A large-scale field study. *ACM Transactions on Storage (TOS)*, 17(1):1–27, 2021. 1, 2, 8
- [45] Ioannis Manousakis, Sriram Sankar, Gregg McKnight, Thu D Nguyen, and Ricardo Bianchini. Environmental conditions and disk reliability in free-cooled datacenters. In *14th USENIX conference on file and storage technologies (FAST 16)*, pages 53–65, 2016. 9
- [46] Pascale Minet, Eric Renault, Ines Khoufi, and Selma Boumerdassi. Analyzing traces from a google data center. In *International Wireless Communications & Mobile Computing Conference*, pages 1167–1172, 2018. 1, 2
- [47] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990. 8
- [48] OpenCompute. Server/projectolympus. <https://www.opencompute.org/wiki/Server/ProjectOlympus> accessed 6/26/2022, November 2017. 1, 2
- [49] Jehan-François Paris, Ahmed Amer, Darrell D. E. Long, and Thomas J. E. Schwarz. Self-repairing disk arrays. arXiv cs.DC 1501.00513, 2015. 8
- [50] Jehan-François Paris, Darrell D.E. Long, and S.J. Thomas Schwarz. Zero-maintenance disk arrays. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 140–141, 2013. 8
- [51] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD '88*, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery. 8
- [52] Borja Peleato, Haleh Tabrizi, Rajiv Agarwal, and Jeffrey Ferreira. Ber-based wear leveling and bad block management for nand flash. In *2015 IEEE International Conference on Communications (ICC)*, pages 295–300, 2015. 2, 8
- [53] Sundar Pichai and Ruth Porati. Alphabet announces fourth quarter and fiscal year 2022 results. [https://abc.xyz/investor/static/pdf/2022Q4\\_alphabet\\_earnings\\_release.pdf?cache=9dela6b](https://abc.xyz/investor/static/pdf/2022Q4_alphabet_earnings_release.pdf?cache=9dela6b) accessed 2/15/23, 2 2023. 2
- [54] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *USENIX FAST*, 2007. 1
- [55] Eric L Pope and Scott P Faasse. Post package repair for mapping to a memory failure pattern, January 2020. US Patent 10,546,649. 2, 5.1, 8
- [56] Meghan Rimol. Gartner predicts half of cloud data centers will deploy robots with ai capabilities by 2025. <https://www.gartner.com/en/newsroom/press-releases/2021-11-01-gartner-predicts-half-of-cloud-data-centers-will-deploy-robots-with-ai-capabilities-by-2025> accessed 2/15/23, 2021. 8
- [57] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *Commun. ACM*, 54(2):100–107, feb 2011. 1
- [58] Fumiyoshi Shoji, Shuji Matsui, Mitsuo Okamoto, Fumichika Sueyasu, Toshiyuki Tsukamoto, Atsuya Uno, and Keiji Yamamoto. Long term failure analysis of 10 peta-scale supercomputer. *HPC in Asia Poster, ISC*, 2015. 1
- [59] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *IEEE SC*, pages 1–11, 2012. 1
- [60] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coiommu: A virtual iommu with cooperative dma buffer tracking for efficient memory management in direct i/o. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 479–492, 2020. 2
- [61] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020. 1, 2
- [62] Kushagra Vaid. Datacenter power efficiency: Separating fact from fiction. In *Invited talk at the 2010 Workshop on Power Aware Computing and Systems*, volume 1, 2010. 2
- [63] Remco Van Erp, Reza Soleimanzadeh, Luca Nela, Georgios Kampitsis, and Elison Matioli. Co-designing electronics with microfluidics for more sustainable cooling. *Nature*, 585(7824):211–216, 2020. 1
- [64] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *ACM EuroSys*, pages 1–17, 2015. 1, 2

- [65] Kashi Venkatesh Vishwanath, Albert Greenberg, and Daniel A. Reed. Modular data centers: How to design them? In *Proceedings of the 1st ACM Workshop on Large-Scale System and Application Performance*, LSAP '09, page 3–10, New York, NY, USA, 2009. Association for Computing Machinery. 8
- [66] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, 2010. 1, 2
- [67] Osamu Wada, Toshimasa Namekawa, Hiroshi Ito, Atsushi Nakayama, and Shuso Fujii. Post-packaging auto repair techniques for fast row cycle embedded dram. In *2004 International Conference on Test*, pages 1016–1023. IEEE, 2004. 2, 5.1, 8
- [68] Chundong Wang and Weng-Fai Wong. Extending the lifetime of nand flash memory by salvaging bad blocks. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 260–263, 2012. 2, 8
- [69] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 25–36, 2017. 1, 2, 2
- [70] Paul Willmann, Scott Rixner, and Alan L Cox. Protection strategies for direct access to virtualized i/o devices. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008. 2
- [71] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the dma mapping problem in direct device assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–12, 2010. 2
- [72] Yangfan Zhong. Experiences with immersion cooling in alibaba datacenter. OpenCompute 2019 track on Rack & Power / Advanced Cooling <https://www.youtube.com/watch?v=GMSLjr7Wlis&t=1067s> accessed 6/26/2022, October 2019. 1, 9
- [73] Ali Zolghadri. A redundancy-based strategy for safety management in a modern civil aircraft. *Control Engineering Practice*, 8(5):545–554, 2000. 8

# NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall

Haonan Lu<sup>\*</sup>, Shuai Mu<sup>†</sup>, Siddhartha Sen<sup>‡</sup>, Wyatt Lloyd<sup>◇</sup>

<sup>\*</sup>University at Buffalo, <sup>†</sup>Stony Brook University,

<sup>‡</sup>Microsoft Research, <sup>◇</sup>Princeton University

## Abstract

Strictly serializable datastores greatly simplify application development. However, existing techniques pay unnecessary costs for naturally consistent transactions, which arrive at servers in an order that is already strictly serializable. We exploit this natural arrival order by executing transactions with minimal costs while optimistically assuming they are naturally consistent, and then leverage a timestamp-based technique to efficiently verify if the execution is indeed consistent. In the process of this design, we identify a fundamental pitfall in relying on timestamps to provide strict serializability and name it the timestamp-inversion pitfall. We show that timestamp inversion has affected several existing systems.

We present Natural Concurrency Control (NCC), a new concurrency control technique that guarantees strict serializability and ensures minimal costs—i.e., one-round latency, lock-free, and non-blocking execution—in the common case by leveraging natural consistency. NCC is enabled by three components: non-blocking execution, decoupled response management, and timestamp-based consistency checking. NCC avoids the timestamp-inversion pitfall with response timing control and proposes two optimization techniques, asynchrony-aware timestamps and smart retry, to reduce false aborts. Moreover, NCC designs a specialized protocol for read-only transactions, which is the first to achieve optimal best-case performance while guaranteeing strict serializability without relying on synchronized clocks. Our evaluation shows NCC outperforms state-of-the-art strictly serializable solutions by an order of magnitude on many workloads.

## 1 Introduction

Strictly serializable datastores have been advocated by much recent work [12, 18, 19, 33, 52, 58, 68] because they provide the powerful abstraction of programming in a single-threaded, transactionally isolated environment, which greatly simplifies application development and prevents consistency anomalies [8]. However, only a few concurrency control techniques provide strict serializability and they are expensive.

Common techniques include distributed optimistic concurrency control (dOCC), distributed two-phase locking (d2PL), and transaction reordering (TR). They incur high overheads which manifest in extra rounds of messages, distributed lock management, blocking, and excessive aborts. The validation round in dOCC, required lock management in d2PL, blocking

during the exchange of ordering information in TR, and aborts due to conflicts in dOCC and d2PL are examples of these four overheads, respectively. These costs are paid to enforce the two requirements of strict serializability: (1) ensuring there is a total order by avoiding interleaving transactions, and (2) ensuring the *real-time ordering* i.e., later-issued transactions take effect after previously-finished ones. However, we find these costs are unnecessary for many *datacenter workloads* where transactions are executed within a datacenter and then replicated within or across datacenters.

Many datacenter transactions do not interleave: e.g., many of them are dominated by reads [12], and the interleaving of reads returning the same value does not affect correctness. Many of them are short [24, 27, 40, 52, 64, 71], and short lifetimes reduce the likelihood of interleaving. Advances in datacenter networking also reduce variance in delivery times of concurrent requests [5, 14, 22], resulting in less interleaving.

In addition, many datacenter transactions arrive at servers in an order that trivially satisfies their real-time order requirement. That is, a transaction arrives at all participant servers after all previously committed transactions.

Because many transactions do not interleave and their arrival order satisfies the real-time order constraints, intuitively, simply executing their requests in the order servers receive them (i.e., treating them as if they were non-transactional simple operations) will naturally satisfy strict serializability. We call these transactions *naturally consistent*.

Ideally, naturally consistent transactions can be safely executed without any concurrency control, incurring zero costs. However, existing techniques pay unnecessary overheads. For instance, dOCC still requires extra rounds of messages for validation, d2PL still acquires locks, and TR still blocks transactions to exchange ordering information, even if validation always succeeds, locks are always available, and nothing needs to be reordered. Therefore, this paper strives to make naturally consistent transactions as cheap as possible.

In this paper, we present Natural Concurrency Control (NCC), a new concurrency control technique that guarantees strict serializability and ensures minimal costs—i.e., one-round latency, lock-free, and non-blocking execution—in the common case. NCC’s design insight is to execute naturally consistent transactions in the order they arrive, as if they were non-transactional operations, while guaranteeing correctness without interfering with transaction execution.



NCC is enabled by three components. *Non-blocking execution* ensures that servers execute transactions in a way that is similar to executing non-transactional operations. *Decoupled response management* separates the execution of requests from the sending of their responses, ensuring that only correct results are returned. *Timestamp-based consistency checking* uses timestamps to verify transactions' results, without interfering with execution.

While designing the consistency-checking component, we identified a correctness pitfall in timestamp-based, strictly serializable techniques. Specifically, these techniques sometimes fail to guard against an execution order that is total but incorrectly inverts the real-time ordering between transactions, thus violating strict serializability. We call this the *timestamp-inversion* pitfall. Timestamp inversion is subtle because it can happen only if a transaction interleaves with a set of *non-conflicting* transactions that have real-time order relationships. The pitfall is fundamental as we find it affects multiple prior systems (TAPIR [71] and DrTM [66]), which, as a result, do not provide strict serializability as claimed.

NCC handles timestamp inversion through response timing control (RTC), an integral part of decoupled response management, without interfering with non-blocking execution or relying on synchronized clocks. NCC proposes two timestamp optimization techniques, asynchrony-aware timestamps and smart retry, to reduce false aborts. Moreover, NCC designs a specialized protocol for read-only transactions, which, to the best of our knowledge, is the first to achieve optimal performance [40] in the best case while ensuring strict serializability, without relying on synchronized clocks.

We compare NCC with common strictly serializable techniques: dOCC, d2PL, and TR, and two serializable protocols, TAPIR [71] and MVTO [55]. We use three workloads: Google-F1, Facebook-TAO, and TPC-C (§6). The Google-F1 and Facebook-TAO workloads synthesize production-like workloads for Google's Spanner [12, 59] and Facebook's TAO [10], respectively. Both workloads are read-dominated. TPC-C [63] consists of few-shot transactions that are write-intensive. We further explore the workload space by varying the write fractions in Google-F1. NCC significantly outperforms dOCC, d2PL, and TR with 2–10× lower latency and 2–20× higher throughput. NCC outperforms TAPIR with 2× higher throughput and 2× lower latency, and closely matches the performance of MVTO.

In summary, this work makes the following contributions:

- Identifies timestamp inversion, a fundamental correctness pitfall in timestamp-based, strictly serializable concurrency control techniques.
- Proposes NCC, a new concurrency control technique that provides strict serializability and achieves minimal overhead in the common case by exploiting natural consistency in datacenter workloads.
- A strictly serializable read-only protocol with optimal best-

case performance that does not rely on synchronized clocks.

- An implementation and evaluation that shows NCC outperforms existing strictly serializable systems by an order of magnitude and closely matches the performance of systems that provide weaker consistency.

## 2 Background

This section provides the necessary background on transactional datastores, strict serializability, and general techniques for providing strict serializability.

### 2.1 Transactional Datastores

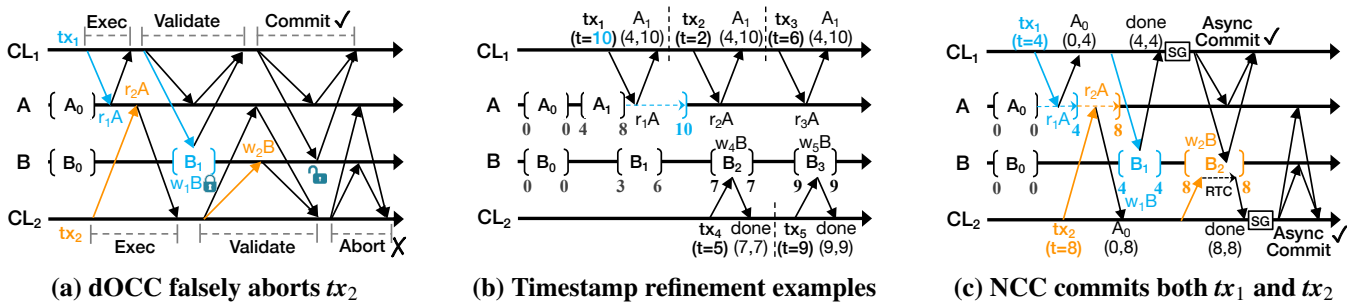
Transactional datastores are the back-end workhorse of many web applications. They typically consist of two types of machines. Front-end *client* machines receive users' requests, e.g., managing a web page, and execute these requests on behalf of users by issuing transactions to the storage *servers* that store the data. Servers are fault-tolerant, e.g., the system state is made persistent on disks and replicated via replicated state machines (RSM), like Paxos [30].

Transactions are managed by coordinators, which can be co-located either with a server or the client. This paper adopts the latter approach to avoid the delays caused by shipping the transaction from the client to a server, while explicitly handling client failures. The coordinator issues read/write operations to relevant servers, called *participants*, following the transaction's logic, which can be *one-shot*, i.e., it knows a priori which data to read/write and can send all requests in one step, or *multi-shot*, i.e., it takes multiple steps as the data read in one step determines which data to read/write in later steps. The system executes transactions following a concurrency control protocol, which ensures that transactions appear to take effect in an order that satisfies the system's consistency requirements. The stronger the consistency provided by the system, the easier it is to develop correct applications.

### 2.2 Strict Serializability

*Strict serializability* [23, 53], also known as external consistency [21], is often considered the strongest consistency model. It requires that (1) there exists a *total order* of transactions, and (2) the total order must respect the *real-time order*, which means if transaction  $tx_1$  ends before  $tx_2$  starts, then  $tx_1$  must appear before  $tx_2$  in the total order. As a result, transactions appear to take effect one at a time in the order the system receives them.

**Formal definition.** We use Real-time Serialization Graphs (RSG) [1] to formalize the total order and real-time order requirements. An RSG is a directed graph that captures the order in which transactions take effect. Specifically, two requests from different transactions have an *execution edge*  $req_1 \xrightarrow{\text{exe}} req_2$  if any of the following happens:  $req_1$  creates some data version  $v_i$  and  $req_2$  reads  $v_i$ ;  $req_1$  reads some data version  $v_j$  and  $req_2$  creates  $v_j$ 's next version that is after  $v_j$ ; or



**Figure 1:**  $tx_1$  and  $tx_2$  are naturally consistent. dOCC incurs unnecessary validation costs, and  $tx_2$  could be falsely aborted due to lock unavailability. NCC can commit both transactions with timestamp pre-assignment, refinement, and the safeguard check (denoted by SG). These techniques are detailed in Section 5.1. Each version in NCC has a  $(t_w, t_r)$  pair which is included in server responses. RTC means response timing control, detailed in Section 5.2.

$req_1$  creates some data version  $v_k$  and  $req_2$  creates  $v$ 's next version that is after  $v_k$ . Two transactions have an execution edge  $tx_1 \xrightarrow{\text{exe}} tx_2$  if there exist  $req_1$  and  $req_2$  from  $tx_1$  and  $tx_2$ , respectively, such that  $req_1 \xrightarrow{\text{exe}} req_2$ . A chain of execution edges constructs a directed path between two transactions (requests), denoted by  $tx_1 \xrightarrow{\text{exe}} tx_2$  ( $req_1 \xrightarrow{\text{exe}} req_2$ ), meaning that  $tx_1$  ( $req_1$ ) “transitively” affects  $tx_2$  ( $req_2$ ) through some intermediary transactions (requests). Two transactions have a *real-time edge*  $tx_1 \xrightarrow{\text{rto}} tx_2$  if there is a real-time ordering between  $tx_1$  and  $tx_2$ , meaning that  $tx_1$  commits before  $tx_2$ 's client issues  $tx_2$ 's first request. In an RSG, vertices are committed transactions, connected by execution and real-time edges.

There exists a total order if and only if transactions do not circularly affect each other. That is, the subgraph that comprises all vertices and only execution edges is acyclic, meaning that the following invariant holds:

**Invariant 1:**  $\forall tx_1, tx_2 (tx_1 \xrightarrow{\text{exe}} tx_2 \implies \neg(tx_2 \xrightarrow{\text{exe}} tx_1))$

The (total) execution order respects the real-time order if and only if the execution edges (paths) do not *invert* the real-time edges, meaning that the following invariant holds:

**Invariant 2:**  $\forall tx_1, tx_2 (tx_1 \xrightarrow{\text{rto}} tx_2 \implies \neg(tx_2 \xrightarrow{\text{exe}} tx_1))$

These invariants correspond to the total order and real-time order requirements, respectively. Therefore, a system is strictly serializable if and only if for any execution it allows, both invariants hold.

By enforcing a total order and the real-time order, strictly serializable systems provide application programmers with the powerful abstraction of programming in a single-threaded, transactionally isolated environment, and thus they greatly simplify application development and eliminate consistency anomalies. For example, if an admin removes Alice from a shared album and then notifies Bob of the change (via a channel external to the system, e.g., a phone call), who then uploads a photo he does not want Alice to see, then Alice must not see Bob's photo, since *remove\_Alice*  $\xrightarrow{\text{rto}}$  *new\_photo*. Such guarantees cannot be enforced by weaker consistency models, e.g., serializability, because they do not enforce the real-time order that is external to the system.

## 2.3 dOCC, d2PL, & Transaction Reordering

Only a few techniques provide strict serializability. The common ones are dOCC, d2PL, and transaction reordering (TR). dOCC and d2PL typically require three round trips, one for each phase: execute, prepare, and commit. In the execute phase, the coordinator reads the data from the servers while writes are buffered locally. d2PL acquires read locks in this phase while dOCC does not. In the prepare phase, the coordinator sends prepare messages and the buffered writes to the participant servers. d2PL locks all participants while dOCC only locks the written data. dOCC must also validate that values read in the execute phase have not changed. If all requests are successfully prepared, i.e., locks are available and/or validation succeeds, the coordinator notifies the participants to commit the transaction and apply the writes; otherwise, the transaction is aborted and retried.

Transaction reordering typically requires two steps. In the first step, the coordinator sends the requests to the servers, which make requests wait while recording their arrival order relative to those of concurrent transactions. This ordering information usually increases linearly in size with respect to the number of concurrent transactions. In the second step, the coordinator collects the ordering information from participants, sorts the requests to eliminate interleavings, and servers execute the transactions in the sorted order.

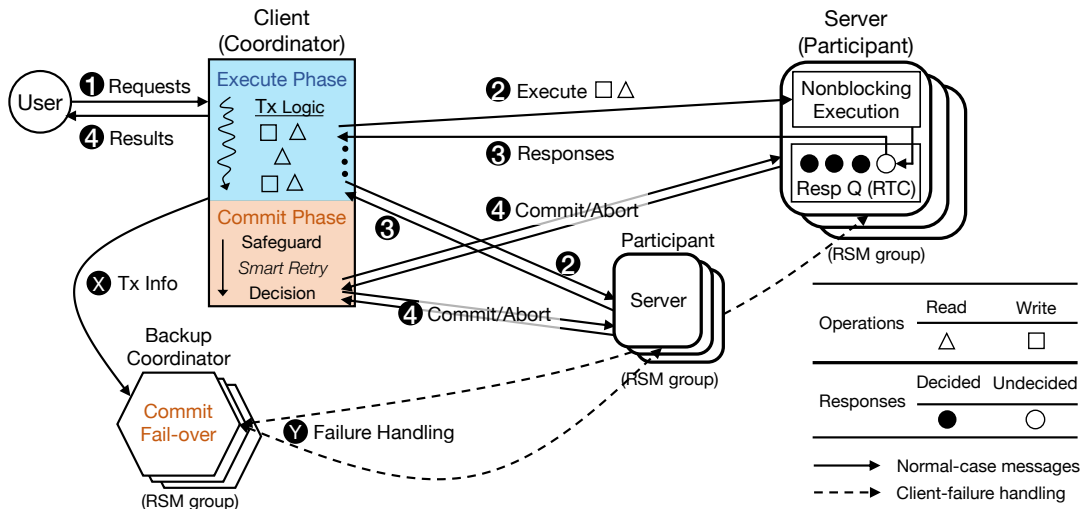
These techniques are expensive, e.g., they require multiple rounds of messages, locking, waiting, and aborts. We find that these overheads are wasteful for most of the transactions in many datacenter workloads, and this observation has inspired our protocol design.

## 3 Design Insight & Overview

This section explains natural consistency, which inspires our design, and overviews the key design components.

### 3.1 Exploiting Natural Consistency

For many datacenter transactions, simply executing their requests in the order servers receive them, as if they were non-transactional read/write operations, would naturally satisfy



**Figure 2: An overview of system architecture and transaction execution. NCC follows two-phase commit and has three design pillars: non-blocking execution, decoupled response management, and timestamp-based consistency checking.**

strict serializability. In other words, they arrive at servers in an order that is already strictly serializable. We call these transactions *naturally consistent*. Key to natural consistency is the arrival order of transaction requests.

Many requests in datacenter workloads arrive in an order that is total, i.e., transactions do not circularly affect each other, due to the following reasons. First, many requests in real-world workloads are reads [10, 12], and reads do not affect other reads. For instance, reads that return the same value can be executed in any order, and thus servers can safely execute them in their arrival order. Second, many transactions are short, e.g., they are one-shot [24, 27, 40, 52, 64, 71] or can be made one-shot using stored procedures [20, 34, 51, 60, 67], and thus their requests are less likely to interleave with others’ requests. Third, advances in datacenter networks reduce the variance of message delivery times [49, 50, 54], and thus further reduces the likelihood of request interleaving.

In most cases, the (total) arrival order satisfies the real-time order between transactions because a transaction that happens later in real-time, i.e., it starts after another transaction has been committed, must arrive at servers after the committed transaction has arrived.

Ideally, the system would treat naturally consistent transactions as non-transactional operations and execute them in the order they arrive without any concurrency control, while still guaranteeing strict serializability. This insight suggests room for improvement in existing techniques. For instance, dOCC still requires validation messages which are unnecessary when transactions are naturally consistent. Further, during validation between prepare and commit, dOCC has a *contention window* where it can cause other concurrent transactions to abort. As shown in Figure 1a, such contention windows lead to *false aborts*, where a transaction is aborted despite being consistent. Our design aims to minimize costs for as many

naturally consistent transactions as possible.

### 3.2 Three Pillars of Design

Our design executes naturally consistent transactions in a manner that closely resembles non-transactional operations. This is made possible through three components.

**Non-blocking execution.** Assuming transactions are naturally consistent, servers execute requests in the order they arrive. Requests are executed “urgently” to completion without acquiring locks, and their results are immediately made visible to prevent blocking subsequent requests. As a result, transactions are executed as cheaply as non-transactional operations, without incurring contention windows.

**Decoupled response management.** Because not all transactions are naturally consistent, servers must prevent returning inconsistent results to clients and ensure there are no cascading aborts. This is achieved by decoupling requests’ responses from their execution, with a response sent asynchronously only once it is verified consistent. Inconsistent results are discarded, and their requests are re-executed.

**Timestamp-based consistency checking.** We must check consistency as efficiently as possible, without interfering with server-side execution. We leverage timestamps to capture the arrival order (thus the execution order) of requests and design a client-side checker that verifies if requests were executed in a total order, without incurring overheads such as messages (as in dOCC and TR) or locks (as in dOCC and d2PL).

Figure 2 shows at a high level how these three pillars support our design, and depicts the life cycle of transactions:

- 1 The user submits application requests to a client, which translates the requests into transactions.
- 2 The (client) coordinator sends operations to the par-

ticipant servers, following the transaction’s logic. The servers execute requests in their arrival order. Their responses are inserted into a queue and sent asynchronously. The responses include timestamps that capture requests’ execution order.

- ③ Responses are sent to the client when it is safe, determined by response timing control (RTC).
- ④ The safeguard checks if transactions were executed in a total order by examining the timestamps in responses. The coordinator sends commit/abort messages to the servers and returns the results of committed transactions to the user in parallel, without waiting for servers’ acknowledgments.  $\otimes$  and  $\otimes$  explicitly handle client failures by leveraging a server as a backup coordinator.

**Limitations.** First, our design leverages natural consistency, which is observed in short (e.g., one or few shots) datacenter transactions; while our design supports arbitrary-shot transactions, many-shot long-lasting transactions that are more likely to interleave might not benefit from our design. Second, the timestamps associated with each request, including both reads and writes, must be made persistent (e.g., written to disks) and replicated for correctly handling failures, which could lead to replication overhead, which we detail in Section 5.6.

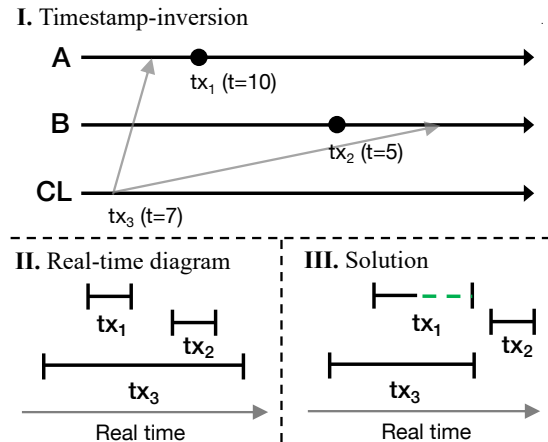
**An observation.** Key to the correctness of our design is leveraging timestamps to verify a total order that respects the real-time order. Yet, we identify a correctness pitfall in relying on timestamps to ensure strict serializability.

## 4 Timestamp-Inversion Pitfall

We discover that timestamp-based techniques sometimes fail to guard against a total order that violates the real-time order in subtle cases. As a result, executing transactions in such a total order inverts the real-time relationship between transactions, which leads to a violation of strict serializability. We call such violations the *timestamp-inversion* pitfall. Figure 3 shows a minimal construction of timestamp inversion using three transactions.  $tx_1$  and  $tx_2$  are single-machine transactions issued by different clients, and  $tx_2$  starts after  $tx_1$  finishes, so there exists a real-time order  $tx_1 \xrightarrow{rto} tx_2$  that strict serializability must enforce.  $tx_3$  is a multi-shard transaction by a third client that interleaves with  $tx_1$  and  $tx_2$ .  $tx_1$ ,  $tx_2$ , and  $tx_3$  have timestamps 10, 5, and 7, respectively.<sup>1</sup> By following these timestamps, the transactions are executed in a total order denoted as  $tx_2 \xrightarrow{exe} tx_3 \xrightarrow{exe} tx_1$ , which inverts the real-time order  $tx_1 \xrightarrow{rto} tx_2$  and thus violates strict serializability. Specifically, the execution of these transactions violates Invariant 2, subjecting them to consistency anomalies discussed in §2.2.

The timestamp-inversion pitfall is subtle because it happens only if a transaction interleaves with a set of *non-conflicting* transactions that have real-time ordering constraints. We find

<sup>1</sup>A timestamp is generated by either a loosely synchronized physical clock [48] or a causal counter, e.g., a Lamport clock [28].



**Figure 3: A minimal example of timestamp inversion, a real-time diagram shows the ordering of transactions, and how NCC tackles the timestamp-inversion pitfall.**

timestamp inversion to be fundamental as it has affected multiple different systems; we discuss two such systems below. In addition, we find that there are several existing systems that do not explicitly define their consistency model, but give a strong indication of providing strict serializability—e.g., they claim invariants that are equivalent to strict serializability, or are built on or evaluated against strictly serializable protocols. We find that these systems also fall into the pitfall.

**Timestamp inversion affects several prior systems.** The minimal example in Figure 3 can be extended to variants of timestamp inversion that affect different types of transactions in real system designs, suggesting that this pitfall is general and fundamental. For instance, we find two systems from recent SOSPs fall into different variants of the pitfall, and thus are not strictly serializable as claimed. We elaborate below to help future work avoid timestamp inversion, and provide the full counterexamples in a technical report [41].

TAPIR [71, 72] is an integrated protocol that co-designs concurrency control and replication. Its concurrency control is a variant of dOCC which validates writes using timestamps without acquiring locks, while reads are validated in the traditional way. Because reads and writes are executed in timestamp order but validated with separate mechanisms, TAPIR’s read-write transactions may cause an inversion of concurrent writes. For instance, if  $tx_1$ ,  $tx_2$ , and  $tx_3$  in Figure 3 are read-write transactions, then all three transactions would pass TAPIR’s validation, which results in the inversion of  $tx_1 \xrightarrow{rto} tx_2$ . The effect of this inversion is perceivable to the client via future reads. This variant of timestamp inversion requires a detailed analysis of the possible executions, showing that none of them are admissible by strict serializability [41].

DrTM [11, 66] is a specialized design for modern datastores equipped with hardware transactional memory and remote direct memory access. DrTM uses timestamps to validate read leases which are acquired before reading the data, a technique equivalent to executing read requests in the timestamp order.



---

**Algorithm 5.1: Client (transaction coordinator) logic**

---

```
1 Function EXECUTERWTRANSACTION(tx) :
2   results ← {}; t_pairs ← {} // server responses
3   t.clk ← ASYNCHRONYWARETS(tx); t.cid ← clientID
4   for req in tx do
5     // send requests shot by shot,
6     // following tx's logic
7     res, t_pair ← NONBLOCKINGEXECUTE(req, t)
8     results ← results ∪ res
9     t_pairs ← t_pairs ∪ t_pair
10
11    // all shots done, tx's logic complete
12    ok, t' ← SAFEGUARDCHECK(t_pairs)
13    if not ok then
14      ok ← SMARTRETRY(tx, t') // §5.4
15
16    if ok then
17      ASYNCCOMMITORABORT(tx, "committed")
18      return results
19
20    else
21      ASYNCCOMMITORABORT(tx, "aborted")
22      go to 2 // abort, and retry from scratch
23
24
25
26
27
```

---

This makes DrTM's read-only transactions subject to inversion, e.g., when  $tx_1$ ,  $tx_2$ , and  $tx_3$  in Figure 3 are read-write, read-write, and read-only transactions, respectively.

The main contributions of TAPIR and DrTM still stand, just with weaker consistency than claimed. Both teams conjecture that they can fix the systems by using synchronized clocks (e.g., TrueTime [12]) and adapting their designs to use these clocks. Thus, it is likely that their contributions still stand with strict serializability when synchronized clocks are used. However, synchronized clocks require specialized infrastructure and are not generally available (§7). Therefore, NCC is designed to avoid timestamp-inversion without relying on synchronized clocks.

## 5 Natural Concurrency Control

This section presents the basic components of NCC, explains how NCC avoids the timestamp-inversion pitfall, introduces two timestamp optimization techniques and a specialized algorithm for read-only transactions, and concludes with discussions of failure handling and correctness.

### 5.1 Protocol Basics

We build NCC on the three design pillars (§3.2) to minimize the costs for naturally consistent transactions.

**Pre-timestamping transactions.** NCC processes a transaction in two phases: execute and commit. Algorithm 5.1 shows the client (coordinator)'s logic. The coordinator starts a transaction  $tx$  by pre-assigning it a timestamp  $t$  that consists of two fields:  $clk$  which is the client's physical time (Section 5.3 details how it is computed), and  $cid$  which is the client identifier.  $t$  uniquely identifies  $tx$  (line 3). When two timestamps have the same  $clk$ , NCC breaks the tie by comparing their  $cid$ .  $t$  is included in all of  $tx$ 's requests that are sent to servers shot by shot, following  $tx$ 's application logic (lines 4 and 5). These timestamps accompany  $tx$  throughout its life cycle and will be used to verify if the results are consistent.

**Refining timestamps to match execution order.** Algorithm 5.2 details the server-side logic for request execution and commitment. Each key stores a list of versions in the order of the server creating them. A version has three fields: *value*, a pair of timestamps  $(t_w, t_r)$ , and *status*. *value* stores the data;  $t_w$  is the timestamp of the transaction that created the version;  $t_r$  is the highest timestamp of transactions that read the version; and *status* indicates the state of the transaction that created the version: either (initially) *undecided*, or *committed*. An aborted version is removed from the datastore.

The server always executes a request against the most recent version  $curr\_ver$ , which is either undecided or committed (line 35). Specifically, the server executes a write by creating a new undecided version  $new\_ver$ , which is now the most recent version of the key, ordered after  $curr\_ver$  (lines 39 and 40), and executes a read by reading the *value* of  $curr\_ver$  (line 44). NCC's basic protocol can work with a single-versioned data store while multi-versioning is required only for smart retry, a timestamp optimization technique (§5.4). The server refines the most recent version's timestamp pair to match the order in which requests are executed. Specifically, a write request computes  $new\_ver$ 's  $t_w$  as follows: its physical time field is no less than that of the write's timestamp  $t$  and that of  $curr\_ver$ 's  $t_r$ , and its client identifier is the same as  $t$ 's (line 37);  $new\_ver$ 's  $t_r$  is initialized to  $t_w$  (line 38). Similarly, a read request updates  $curr\_ver$ 's  $t_r$  if needed (line 43). Figure 1b shows examples of how timestamps are refined. A version is associated with a  $t_w$  and a  $t_r$ , e.g.,  $A_1$  initially has a timestamp pair (4, 8).  $tx_1$ – $tx_3$  are single-key read transactions with pre-assigned timestamps 10, 2, and 6, respectively. They return the most recent version of  $A$ , i.e.,  $A_1$ , update its  $t_r$  if needed, and return  $A_1$ 's timestamp pair.  $tx_4$  and  $tx_5$  show how writes manage timestamps.

These (refined) timestamps match requests' arrival order and thus also match the execution order: on each key, a read must have a timestamp greater than that of the write it sees, i.e., a read is ordered after the most recent write, and a write must have a timestamp greater than that of the most recent read, i.e., a write is ordered after the most recent read (and thus all previous writes).

**Non-blocking execution and response queues.** The server executes requests in a non-blocking manner and decouples

---

**Algorithm 5.2: Server execution and commitment**

---

```
28 Multi-versioned data store:
29 DS[key][ver] // indexed by key, vers sorted by  $t_w$ 
// ver is either committed or undecided
30 Response queue:
31 resp_qs[key][resp_q] // resp queues for each key
32
33 Function NONBLOCKINGEXECUTE(req, t) :
34   resp ← [] // response message
35   curr_ver ← DS[req.key].most_recent
36   if req is write then
37      $t_w.clk ← \max\{t.clk, curr\_ver.t_r.clk + 1\}; t_w.cid ← t.cid$ 
38      $t_r ← t_w$ 
39     new_ver ← [req.value, ( $t_w, t_r$ ), "undecided"]
40     DS[req.key] ← DS[req.key] + new_ver
41     resp ← ["done", ( $t_w, t_r$ )]
42   else
43      $curr\_ver.t_r ← \max\{t, curr\_ver.t_r\}$ 
44     resp ← [curr_ver.value, ( $curr\_ver.t_w, curr\_ver.t_r$ )]
45   resp_qs[req.key].enqueue(resp, req, t, "undecided")
46   RESPTIMINGCONTROL(resp_qs[req.key]) // §5.2
47
48 Function ASYNCCOMMITORABORT(tx, decision) :
49   foreach ver created by tx do
50     if decision = "committed" then
51       ver.status ← decision
52     else
53       DS.remove(ver)
54   foreach resp_q in resp_qs do
55     foreach resp in resp_q do
56       if resp.request ∈ tx then
57         resp.q_status ← decision
58   RESPTIMINGCONTROL(resp_q) // §5.2
```

---

their execution from responses. Specifically, a write creates a version and immediately makes it visible to subsequent transactions; a read fetches the value of the most recent version whose *status* could be undecided, without waiting for it to commit; the server prepares the response (lines 34, 41, and 44), inserts it into a *response queue* (lines 45 and 46), which asynchronously sends the responses to clients when it is safe. (Section 5.2 details response timing control, which determines when sending a response is safe so timestamp inversion and cascading aborts are prevented.) Unlike d2PL and dOCC, which lock data for at least one round-trip time in the execute and prepare phases (i.e., the contention window), non-blocking execution ensures that a transaction never exclusively owns the data without performing useful work. As a result, the server never stalls, and CPUs are fully utilized to execute requests. Moreover, non-blocking execution eliminates the contention window and thus reduces false aborts.

**Client-side safeguard.** A server response includes the timestamp pair ( $t_w, t_r$ ) of the most recent version, e.g., *new\_ver* for a write and *curr\_ver* for a read. The returned ( $t_w, t_r$ ) represents the time range in which the request is valid. That is, a read must take effect after  $t_w$ , which is the time when the

most recent write on the same key took effect, and no later writes can take effect between  $t_w$  and  $t_r$  on the same key. A write must have  $t_w = t_r$ , meaning that it takes effect exactly at  $t_w$ . When a transaction has completed its logic (i.e., all shots are executed) and the client has received responses to all its requests, the safeguard looks for a consistent snapshot that intersects all ( $t_w, t_r$ ) pairs in server responses by checking if the ( $t_w, t_r$ ) pairs overlap (lines 8, 18–27). This intersecting snapshot identifies the transaction’s synchronization point, i.e., all requests are valid at the intersecting timestamp.

Figure 1c shows an example where NCC executes the same transactions in Figure 1a. The default versions  $A_0$  and  $B_0$  both have a timestamp pair (0, 0).  $tx_1$  and  $tx_2$  are pre-assigned 4 and 8, respectively, and their requests arrive in the same order as they were in Figure 1a. The safeguard enables NCC to commit both transactions, i.e.,  $tx_1$ ’s responses intersect at 4 while  $tx_2$ ’s intersect at 8, without unnecessary overhead such as dOCC’s validation cost and false aborts.

When the client has decided to commit or abort the transaction, the protocol enters the commit phase by sending the commit/abort messages to the servers. If the transaction is committed, the server updates the *status* of the created versions from undecided to committed; otherwise, the versions are deleted (lines 48–53). The client retries the aborted transaction. The client sends the results of the committed transaction to the user in parallel with the commit messages, i.e., asynchronous commit, without waiting for servers’ acknowledgments (lines 11–16).

**Supporting complex transaction logic.** NCC supports transactions accessing a key multiple times, e.g., read-modify-writes and repeated reads/writes, by treating its requests to the same key as a single logical request. For instance, if a read-modify-write has its read and write requests executed consecutively (i.e., they are not intersected by other writes), then only the write response is checked by the safeguard, treating read-modify-write as one logical request; otherwise, it is aborted if there are intersecting writes, e.g., when the most recent version has a  $t_w$  greater than that returned by the read of this read-modify-write. The responses of these requests are grouped together in the response queue, e.g., the write response of a read-modify-write is inserted right after the read response of the same read-modify-write. We explain the details of handling complex logic in the technical report [41].

NCC achieves minimal costs by urgently executing transactions in a non-blocking manner and by ensuring a total order with the light-weight timestamp-based safeguard. Yet, in order to provide strict serializability, NCC must enforce the real-time order between transactions by handling the timestamp-inversion pitfall, as we discuss next.

## 5.2 Response Timing Control

NCC avoids the timestamp-inversion pitfall by disentangling the subtle interleaving between a set of non-conflicting transactions that have real-time order dependencies (e.g., Figure 3),

without relying on synchronized clocks. Specifically, NCC introduces *response timing control* (RTC), which controls the sending time of responses. It is safe to send the response of a request  $req_1$  when the following dependencies are satisfied:

- D<sub>1</sub> If  $req_1$  reads a version created by  $req_0$  of another transaction, then  $req_1$ 's response is not returned until  $req_0$  is committed or it is discarded if  $req_0$  is aborted (then  $req_1$  will be re-executed).
- D<sub>2</sub> If  $req_1$  is a write and there are reads that read the version which immediately precedes the one created by  $req_1$ , then  $req_1$ 's response is not returned until the reads are committed/aborted.
- D<sub>3</sub> If  $req_1$  creates a version immediately after the version created by  $req_0$  of another transaction, then  $req_1$ 's response is not returned until  $req_0$  is committed/aborted.

By enforcing these dependencies, NCC controls the sending of responses so that the transactions which form the subtle interleaving are forced to take effect in their real-time order. For instance, in Figure 3, server A cannot send the response of  $tx_1$  until  $tx_3$  has been committed (assuming at least one of them writes to A). As a result, any transaction  $tx_2$  that begins after  $tx_1$  receives its response, i.e.,  $tx_1 \xrightarrow{rto} tx_2$ , must be executed after  $tx_1$ , and thus after  $tx_3$  as well:  $tx_2$ 's execution on each server is after it begins, which is after  $tx_1$  ends, which is after  $tx_1$ 's response is sent, which is after  $tx_3$  commits, which is after  $tx_3$  executes on each server. This results in a total order  $tx_3 \xrightarrow{exe} tx_1 \xrightarrow{exe} tx_2$ , which respects the real-time order, enforcing Invariant 2, as shown in Part III of Figure 3.

NCC implements RTC by managing response queues, independently from request execution. NCC maintains one queue per key. A queue item consists of four fields: *response* that stores the response message of a request, the *request* itself, *ts* which is the pre-assigned timestamp of the request, and *q\_status* that indicates the state of the request, which is initially *undecided*, and updated to either *committed* or *aborted* when the server receives the commit/abort message for this request (lines 54–57, Algorithm 5.2).

**Managing response queues.** Algorithm 5.3 details how NCC manages the response queue of each key. This logic is invoked every time the server finishes executing a request (line 46) and receives a commit/response message (line 58). NCC iterates over the queue items from the head (i.e., the oldest response) until it finds the first response whose *q\_status* is undecided, which means all earlier requests on the same key have been committed or aborted, i.e., this response has satisfied the three dependencies (lines 60–62 and 71). The server sends this response message to the client if it has not done so (lines 72, 74–77). If this is a read response, then the server sends all consecutive read responses that follow it (lines 73 and 78–81), because all these read responses satisfy the three dependencies. In other words, reads returning the same value do not have dependencies between them. RTC is *effectively* similar to locking the response queues, e.g., the

**Algorithm 5.3:** Response timing control

```

59 Function RESPIMINGCONTROL(resp_q) :
60   head ← resp_q.head() // the oldest response
61   while head.q_status ≠ “undecided” do
62     // find the first response we can send
63     resp_q.dequeue()
64     new_head ← resp_q.head()
65     new_req ← new_head.request; t ← new_head.ts
66     while head.q_status = “aborted”
67       and head.request is write and new_req is read do
68         // handle reads seeing aborted writes
69         resp_q.dequeue() // discard read response
70         // re-execute the read locally
71         NONBLOCKINGEXECUTE(new_req, t)
72         new_head ← resp_q.head()
73         new_req ← new_head.request; t ← new_head.ts
74     head ← resp_q.head()
75   curr_item ← head
76   repeated loop
77     // send dependency-satisfied responses
78     resp ← curr_item.response
79     if resp.is_sent ≠ true then
80       sys_call.send(resp) // send to client
81       resp.is_sent ← true
82     // send consecutive read responses
83     next_item ← curr_item.next()
84     if curr_item.request is not read
85     or next_item.request is not read then
86       break repeated loop
87     curr_item ← next_item

```

queue is “locked” when a response is sent and other responses must wait, and is “unlocked” when the commit/abort message for the request to which the sent response belongs is received. However, RTC differs from lock-based mechanisms in that it is decoupled from execution and does not introduce contention windows, i.e., data objects are not locked.

**Fixing reads locally.** When the server receives an abort message for a write request, it must invalidate the responses of any reads that have fetched the value of the aborted write. This is necessary to avoid returning invalid results to the client and to prevent cascading aborts. Specifically, the server removes the response of such a read from the response queue and re-executes the read request, e.g., it fetches the current most recent version, prepares a new response, and inserts the new response to the tail of the queue (lines 65–68).

**Avoiding indefinite waits.** To avoid responses from circularly waiting on dependencies across different keys, NCC early aborts a request (thereby aborting the transaction to which it belongs) if its pre-assigned timestamp is not the highest the server has seen *and* if its response cannot be sent immediately, i.e., it is not the head of the queue. Specifically, a write (read) is aborted if there is an undecided request (write request) with a higher timestamp. Then, the server sends a special response to the client without executing the request. The special response includes a field *early\_abort* which allows

the client to bypass the safeguard and abort the transaction. We omit the details from the pseudocode for clarity.

RTC is a general solution to timestamp inversion, without the need for synchronized clocks. It does not incur more aborts even when responses are not sent immediately, because response management is decoupled from request execution. That is, whether a transaction is committed or aborted is solely based on timestamps, and RTC does not affect either pre-assignment or refinement of timestamps. Yet, NCC’s performance also depends on how well timestamps capture the arrival order of (naturally consistent) transactions. That is, timestamps that do not match transactions’ arrival order could cause transactions to falsely abort even if they are naturally consistent. Next, we will discuss optimization techniques that enable timestamps to better match the arrival order.

### 5.3 Asynchrony-Aware Timestamps

NCC proposes two optimizations: a proactive approach that controls how timestamps are generated before transactions start, and a reactive approach that updates timestamps to match the naturally consistent arrival order after requests are executed. This subsection discusses the proactive approach.

The client pre-assigns the same timestamp to all requests of a transaction; however, these requests may arrive at their participant servers at different physical times, which could result in a mismatch between timestamp and arrival order, as shown in Figure 4a. Transactions  $tx_1$  and  $tx_2$  start around the same time and thus are assigned close timestamps, e.g.,  $t_1 = 1004$  and  $t_2 = 1005$ , respectively (client IDs are omitted). Because the latency between  $B$  and  $CL_1$  is greater than that between  $B$  and  $CL_2$ ,  $tx_1$  may arrive at  $B$  later than  $tx_2$ , but  $tx_1$  has a smaller timestamp. As a result, the safeguard may falsely reject  $tx_1$ , e.g., server  $B$  responds with a refined timestamp pair (1006, 1006) which does not overlap with (1004, 1004), the timestamp pair returned by server  $A$ . However, aborting  $tx_1$  is unnecessary because  $tx_1$  and  $tx_2$  are naturally consistent.

To tackle this challenge, NCC generates timestamps while accounting for the time difference,  $t_\Delta$ , between when a request is sent by the client and when the server starts executing the request. Specifically, the client records the physical time  $t_c$  before sending the request to the server; the server records the physical time  $t_s$  before executing the request and piggy-backs  $t_s$  onto the response sent back to the client; and the client calculates  $t_\Delta$  by finding the difference between  $t_c$  and  $t_s$ , i.e.,  $t_\Delta = t_s - t_c$ . By measuring the end-to-end time difference,  $t_\Delta$  effectively masks the impact of queuing delays and clock skew. The client maintains a  $t_\Delta$  for each server it has contacted. An asynchrony-aware timestamp is generated by adding the client’s current physical time and the greatest  $t_\Delta$  among the servers this transaction will access. For instance, given the values of  $t_\Delta$  shown in Figure 4a,  $CL_1$  assigns  $tx_1$  timestamp 1014 (i.e.,  $1004 + 10$ ) and  $CL_2$  assigns  $tx_2$  1010 (i.e.,  $1005 + 5$ ), and both transactions may successfully pass their safeguard check, capturing natural consistency.

---

#### Algorithm 5.4: Smart retry

---

```

83 Function SMARTRETRY( $tx, t'$ ):
84   foreach  $ver$  accessed by  $tx$  do
85     // next version of the same key
86      $next\_ver \leftarrow ver.next()$ 
87     if  $next\_ver.t_w \leq t'$  then
88       return false
89     if  $ver$  created by  $tx$  and  $ver.t_w \neq ver.t_r$  then
90       return false
91     if  $ver$  created by  $tx$  then
92        $ver.t_w \leftarrow t'; ver.t_r \leftarrow t'$ 
93     else
94        $ver.t_r \leftarrow \max\{ver.t_r, t'\}$ 
95   return true

```

---

### 5.4 Smart Retry

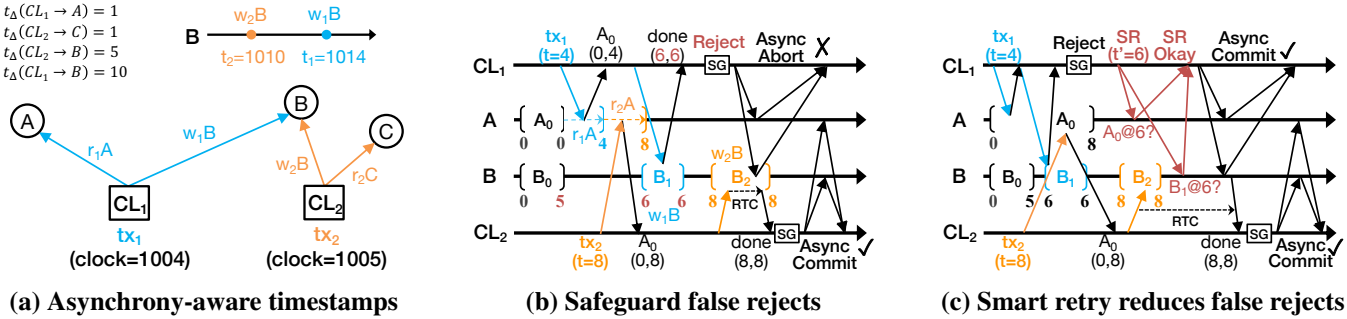
NCC proposes a reactive approach to minimizing the performance impact of the safeguard’s false rejects, which happen when timestamps fail to identify the naturally consistent arrival order, as shown in Figure 4b. Initially, version  $A_0$  has a timestamp pair (0, 0), and  $B_0$  has (0, 5). The same transactions  $tx_1$  and  $tx_2$  as those in Figure 1c access both keys. Following NCC’s protocol,  $tx_1$ ’s responses contain the timestamp pairs (0, 4) and (6, 6) from  $A$  and  $B$ , respectively, which will be rejected by the safeguard because they do not overlap. However, aborting  $tx_1$  is unnecessary because  $tx_1$  and  $tx_2$  are naturally consistent.

Instead, NCC tries to “reposition” a rejected transaction with respect to the transactions before and after it to construct a total order, instead of aborting and re-executing the rejected transaction from scratch, which would waste all the work the server has done for executing it. Specifically, NCC chooses a timestamp that is nearest “in the future” and hopes the rejected transaction can be re-committed at that time. This is possible if the chosen time has not been taken by other transactions.

Algorithm 5.4 shows the pseudocode for smart retry. When the transaction fails the safeguard check, NCC suggests a new timestamp  $t'$ , which is the maximum  $t_w$  in the server responses. The client then sends smart retry messages that include  $t'$  to the participant servers, which then attempt to reposition the transaction’s requests at  $t'$ . The server can reposition a request if there has not been a newer version that was created before  $t'$  (lines 85–87) and, if the request is a write, the version it created has not been read by any transactions (lines 88 and 89). The server updates the timestamps of relevant versions if smart retry succeeds, e.g., the created version has a new timestamp pair ( $t', t'$ ), and  $t_r$  of the read version is updated to  $t'$  if  $t'$  is greater (lines 90–93). (Our implementation does not smart-retry the request that returned the maximum  $t_w$ , i.e.,  $t_w = t'$ , because its smart retry always succeeds.) The client commits the safeguard-rejected transaction if all its smart retry requests succeed, and aborts and retries it from scratch otherwise (lines 9 and 10, Algorithm 5.1).



$$\begin{aligned}
t_{\Delta}(CL_1 \rightarrow A) &= 1 \\
t_{\Delta}(CL_2 \rightarrow C) &= 1 \\
t_{\Delta}(CL_2 \rightarrow B) &= 5 \\
t_{\Delta}(CL_1 \rightarrow B) &= 10
\end{aligned}$$



(a) Asynchrony-aware timestamps (b) Safeguard false rejects (c) Smart retry reduces false rejects

**Figure 4: Optimizations that match the timestamps with transactions' arrival order. Asynchrony-aware timestamps proactively controls the pre-assigned timestamps before execution. Smart retry reactively fixes the safeguard's false rejects after execution thus avoids aborting and re-executing transactions.**

Not only does smart retry avoid false aborts, it also unleashes a higher degree of concurrency, as shown in Figure 4c. The servers have executed a newer transaction  $tx_2$  when  $tx_1$ 's smart retry (SR) messages arrive, and both transactions can be committed even if the messages interleave, e.g.,  $tx_1$ 's smart retry succeeds and  $tx_2$  passes its safeguard check, because  $tx_2$ 's pre-assigned timestamps have left enough room for repositioning  $tx_1$ 's requests. In contrast, validation-based techniques would unnecessarily abort  $tx_1$  (considering SR as dOCC's validation messages) due to the presence of the conflicting transaction  $tx_2$ .

**Garbage collection.** Old versions are temporarily stored and garbage collected as soon as they are no longer needed by undecided transactions for smart retry. Only the most recent versions are used to serve new transactions.

## 5.5 Read-Only Transactions

NCC designs a specialized read-only transaction protocol for read-dominated workloads [10, 12, 26, 40, 44]. Similar to existing works, NCC optimizes read-only transactions by eliminating their commit phase because they do not modify the system state and have nothing to commit. By eliminating commit messages, read-only transactions achieve *optimal performance* in the best case, i.e., one round of non-blocking messages with constant metadata [40, 42, 43].

Eliminating commit messages brings a new challenge to response timing control: write responses can no longer track their dependencies on preceding read-only transactions, as they do not know if and when those reads are committed/aborted. To tackle this challenge, NCC aborts a read-only transaction if it could possibly cause the subtle interleaving that leads to timestamp inversion. In other words, NCC commits a read-only transaction if its requests arrive in a naturally consistent order and no intervening writes have been executed since the last time the client accessed these servers.

Specifically, each client tracks  $t_{ro}$  which is the  $t_w$  of the version created by the most recent write on a server, and the client maintains a map of  $t_{ro}$  for each server this client has contacted. A read-only transaction is identified by a Boolean field

*IS\_READ\_ONLY*. The client sends each of its requests to the participant server together with the pre-assigned timestamp (as in the basic protocol) and the  $t_{ro}$  of the server. To execute a read request, the server checks the version at  $t_{ro}$ . If the version is still the most recent, the server continues to execute the read following the basic protocol, e.g., it fetches the most recent version, refines its  $t_r$  if needed, and returns its timestamp pair; otherwise, the server sends a special response that contains a field *ro\_abort* immediately without executing the request. If any of the responses contain *ro\_abort*, the client aborts this read-only transaction; otherwise, the client continues with the safeguard check and, if needed, smart retry, after which the client does not send any commit/abort messages.

This protocol pays more aborts in the worst case in exchange for reduced message overhead in the normal case, a trade-off that is worthwhile for read-dominated workloads where writes are few so aborts are rare, and read-only transactions are many so the savings in message cost are significant. This protocol also expedites the sending of responses for read-write transactions because read-only transactions do not insert responses into the response queue, i.e., a write response depends only on the reads of preceding read-write transactions in Dependency  $D_2$ , not those of read-only transactions.

## 5.6 Failure Handling

**Tolerating server failures.** NCC assumes servers never fail as their state is typically made persistent on disks and replicated via state machine replication such as Paxos [29]. All state changes incurred by a transaction in the execute phase (e.g.,  $t_w$  and  $t_r$  of each request) must be written to the disk and replicated for correctness. For instance, after a request is executed, the server inserts its response into the response queue and, in parallel, writes the state changes to the disk and replicates the request to other replicas. Its response is sent back to the client when it is allowed by response timing control and when its replication is finished. Commit/abort and smart retry messages are also made persistent and replicated. This simple scheme ensures correctness but incurs high overhead. We plan to investigate possible optimizations in future work, e.g., NCC could defer disk writes and replication to the

last shot of a transaction where all state changes are made persistent and replicated once and for all, without having to replicate each request separately. Server replication inevitably increases latency but does not introduce more aborts, because whether a transaction is committed or aborted is solely based on its timestamps, which are decided during request execution and before replication starts.

**Tolerating client failures.** NCC must handle client failures explicitly because clients are not replicated in most systems and NCC co-locates coordinators with clients. NCC adopts an approach similar to that in Sinfonia [4] and RIFL [31]. We briefly explain it as follows. For a transaction  $tx$ , one of the storage servers  $tx$  accesses is selected as the backup coordinator, and the other servers are cohorts. In the last shot of the transaction logic, which is identified by a field `IS_LAST_SHOT` in the requests, the client notifies the backup coordinator of the identities of the complete set of cohorts. Cohorts always know which server is the backup coordinator. When the client crashes, e.g., is unresponsive for a certain amount of time, the backup coordinator reconstructs the final state of  $tx$  by querying the cohorts for how they executed  $tx$ , and commits/aborts  $tx$  following the same safeguard and smart retry logic. Because computation is deterministic, the backup coordinator always makes the same commit/abort decision as the client would if the client did not fail. To tolerate one client failure, NCC needs one backup coordinator which is a storage server replicated in a usual way.

## 5.7 Correctness

This section provides proof intuition for why NCC is safe and live. At a high level, NCC guarantees a total order, the real-time order, and liveness, with the mechanisms (M<sub>1</sub>) the safeguard, (M<sub>2</sub>) non-blocking execution with response timing control, and (M<sub>3</sub>) early aborts, respectively. We provide a formal proof of correctness in a technical report [41].

**NCC is safe.** We prove that NCC guarantees strict serializability by demonstrating that both Invariants 1 and 2 are upheld. These two invariants correspond to the total order and real-time order requirements, respectively.

Intuitively, NCC commits all requests of a transaction at the same synchronization point, which is the intersection of all  $(t_w, t_r)$  pairs in responses, and the synchronization points of all committed transactions construct a total order. Specifically, we prove that the safeguard enforces Invariant 1, by contradiction. Assume both  $tx_1$  and  $tx_n$  are committed, and  $tx_1 \xrightarrow{\text{exe}} tx_n \xrightarrow{\text{exe}} tx_1$ . Without loss of generality, there must exist a chain of transactions such that  $tx_1 \xrightarrow{\text{exe}} tx_2 \xrightarrow{\text{exe}} \dots \xrightarrow{\text{exe}} tx_n \xrightarrow{\text{exe}} tx_1$ . Then, each transaction may have two requests,  $req$  and  $req'$ , such that  $req'_1 \xrightarrow{\text{exe}} req_2$ ,  $req'_2 \xrightarrow{\text{exe}} req_3$ ,  $\dots$ ,  $req'_{n-1} \xrightarrow{\text{exe}} req_n$ ,  $req'_n \xrightarrow{\text{exe}} req_1$ . Consider their returned timestamps, we can derive the following:

- ①  $t'_{r1} \leq t_{w2}, t'_{r2} \leq t_{w3}, \dots, t'_{rn} \leq t_{w1}$ , by NCC's protocol.
- ②  $t_{w1} \leq t'_{r1}, t_{w2} \leq t'_{r2}, \dots, t_{wn} \leq t'_{rn}$ , because all transactions

are committed and by the safeguard logic.

- ③  $t_{w1} \leq t'_{r1} \leq t_{w2} \leq t'_{r2} \leq \dots \leq t_{wn} \leq t'_{rn} \leq t_{w1}$ , by ① and ②.
- ④  $t'_{r1} = t_{w2} = t'_{r2} = t_{w3} = \dots = t_{wn} = t'_{rn} = t_{w1}$ , by ③.
- ⑤  $req'_i$  is a write and  $req_i$  is a read,  $i \in [1, n]$ , by ④, NCC's protocol, and  $tx_1 \xrightarrow{\text{exe}} tx_2 \xrightarrow{\text{exe}} \dots \xrightarrow{\text{exe}} tx_n \xrightarrow{\text{exe}} tx_1$ .
- ⑥  $t_{w2} = t'_{w1}$  and  $t_{w1} = t'_{wn}$ , by ⑤ and NCC's protocol.
- ⑦  $t'_{w1} = t'_{wn}$ , by ④ and ⑥, which contradicts that writes from different transactions must have distinct  $t_w$  because timestamps are unique. Therefore, Invariant 1 holds.

We prove that NCC enforces Invariant 2 by considering two cases while assuming  $tx_1 \xrightarrow{\text{ro}} tx_2$ . In case 1,  $tx_1$  and  $tx_2$  access some common data items. Then, we must have  $tx_1 \xrightarrow{\text{exe}} tx_2$ , because NCC executes requests in their arrival order. Then, it must be true that  $\neg(tx_2 \xrightarrow{\text{exe}} tx_1)$ , by Invariant 1. In case 2,  $tx_1$  and  $tx_2$  access disjoint data sets, and we prove the claim by contradiction. Assume  $tx_2 \xrightarrow{\text{exe}} tx_1$ , then there must exist  $req_2$  and  $req_1$  in  $tx_2$  and  $tx_1$ , respectively, such that  $req_2 \xrightarrow{\text{exe}} req_1$ .  $req_1$ 's response is not returned until  $req_2$  is committed or aborted, by applying response timing control transitively (§5.2). Then,  $req_2$  is issued before  $req_1$ 's client receives  $req_1$ 's response because a request, e.g.,  $req_2$ , can be committed or aborted only after it is issued and executed. Thus, we can derive  $\neg(tx_1 \xrightarrow{\text{ro}} tx_2)$  because  $tx_2$  has at least one request, e.g.,  $req_2$ , which starts before  $tx_1$  receives all its responses. This means  $tx_2$  starts before  $tx_1$  is committed, which contradicts our assumption  $tx_1 \xrightarrow{\text{ro}} tx_2$ . Therefore, Invariant 2 must hold.

**NCC is live.** NCC's non-blocking execution guarantees that requests always run to completion, i.e., execution never stalls (§5.1). Blocking can happen only to the sending of responses due to response timing control, and NCC avoids circular waiting with early aborts (§5.2). Thus, NCC guarantees that transactions finish eventually.

NCC's specialized read-only transaction protocol and optimization techniques such as asynchrony-aware timestamps and smart retry do not affect correctness, because transactions are protected by the three mechanisms (i.e., M<sub>1</sub>, M<sub>2</sub>, and M<sub>3</sub> summarized at the beginning of this subsection) regardless of whether optimizations or the specialized protocol are used.

## 6 Evaluation

This section answers the following questions:

1. How well does NCC perform, compared to common strictly serializable techniques dOCC, d2PL, and TR?
2. How well does NCC perform, compared to state-of-the-art serializable (weaker consistency) techniques?
3. How well does NCC recover from client failures?

**Implementation.** We developed NCC on Janus's framework [52]. We improved the framework by making it support multi-shot transactions, optimizing its baselines, and adding more benchmarks. NCC's core protocols have  $\sim 3$  K lines of C++ code. We also show the results of NCC-RW, a version

| Workload     | Write fraction   | Assoc-to-obj   | # keys/RO       | # keys/RW           | Value size         | # cols/key     | Zipfian       |
|--------------|------------------|----------------|-----------------|---------------------|--------------------|----------------|---------------|
| Google-F1    | 0.3% [0.3%–30%]  | —              | 1–10            | 1–10                | 1.6KB ± 119B       | 10             | 0.8           |
| Facebook-TAO | 0.2%             | 9.5 : 1        | 1–1K            | 1                   | 1–4KB              | 1–1K           | 0.8           |
| TPC-C        | <b>New-Order</b> | <b>Payment</b> | <b>Delivery</b> | <b>Order-Status</b> | <b>Stock-Level</b> | <b>Dist/WH</b> | <b>WH/svr</b> |
|              | 44%              | 44%            | 4%              | 4%                  | 4%                 | 10             | 8             |

**Figure 5: Workload parameters. RO and RW mean read-only and read-write transactions, respectively. TPC-C has a scaling factor of 10 districts per warehouse and 8 warehouses per server.**

| Workload     | Contention  | # shots    | Characteristics | NCC takeaway                                                |
|--------------|-------------|------------|-----------------|-------------------------------------------------------------|
| Facebook-TAO | Low         | 1          | Read-dominated  | Performance-optimal reads by the RO protocol                |
| Google-F1    | Low         | 1          | Read-dominated  | Performance-optimal reads by the RO protocol                |
| TPC-C        | Medium→High | Multi-shot | Write-intensive | Leverages the natural arrival order, minimizes false aborts |
| Google-WF    | Low→High    | 1          | Write-intensive | Leverages the natural arrival order, minimizes false aborts |

**Figure 6: Facebook-TAO and Google-F1 have low contention. TPC-C and Google-WF (varying write fractions) are write-intensive. TPC-C Payment and Order-Status are multi-shot.**

without the read-only transaction protocol, i.e., all transactions are executed as read-write transactions.

**Baselines.** The evaluation includes three strict serializable baselines (dOCC, d2PL, and Janus) and two serializable baselines (MVTO and TAPIR). We chose d2PL and dOCC because they are the most common strictly serializable techniques. We chose Janus because it is the only open-source TR-based strictly serializable system we could find. We chose MVTO because it has the highest best-case performance among all (weaker) serializable techniques, presenting a performance upper bound. We chose TAPIR because it utilizes timestamp-based concurrency control.

Our evaluation focuses on concurrency control and assumes servers never fail. Janus and TAPIR are unified designs of the concurrency control and replication layers, so we disabled their replication and only compare with their concurrency control protocols, shown as Janus-CC and TAPIR-CC, to make the comparisons fair. We compare with two variants of d2PL. d2PL-no-wait aborts a transaction if the lock is not available. d2PL-wound-wait makes the transaction wait if it has a larger timestamp and aborts the lock-holding transaction otherwise. All baselines are fully optimized: we co-locate coordinators with clients (even if baselines cannot handle client failures), combine the execute and prepare phases for d2PL-no-wait and TAPIR-CC, and enable asynchronous commitment, i.e., the client replies to the user without waiting for the acknowledgments of commit messages.

## 6.1 Workloads and Experimental Setup

We evaluate NCC under three workloads that cover both read-dominated “simpler” transactions and many-write more “complex” transactions. Google-F1 and Facebook-TAO synthesize real-world applications and capture the former: they are one-shot and read-heavy. TPC-C has multi-shot transactions and

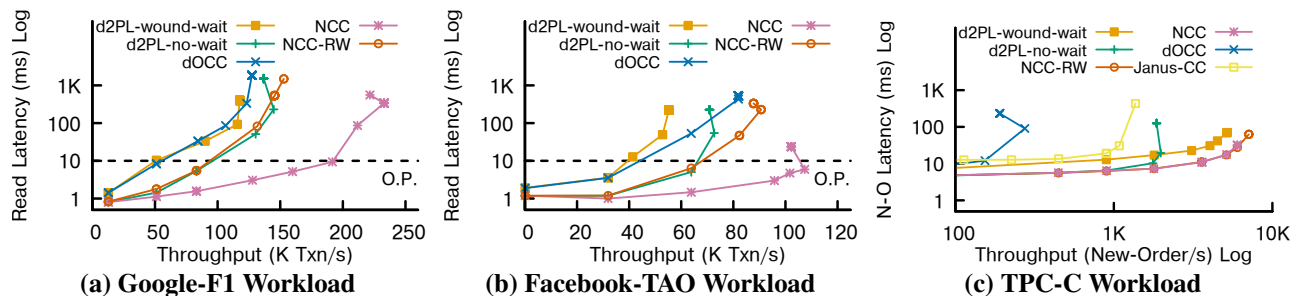
is write-intensive, capturing the latter. We also vary write fractions in Google-F1 to further explore the latter. Table 5 shows the workload parameters.

Google-F1 parameters were published in F1 [59] and Spanner [12]. Facebook-TAO parameters were published in TAO [10]. TPC-C’s New-Order, Payment, and Delivery are read-write transactions. Its Order-Status and Stock-Level are read-only. Janus’s original implementation of TPC-C is one-shot, so we modified it to make Payment and Order-Status multi-shot, to demonstrate NCC is compatible with multi-shot transactions and evaluate its performance beyond one-shot transactions (though they are still relatively short).

**Experimental setting.** We use Microsoft Azure [47]. Each machine has 4 CPUs (8 cores), 16GB memory, and a 1Gbps network interface. We use 8 machines as servers and 16–32 machines as clients that generate open-loop requests to saturate the servers. (The open-loop clients back off when the system is overloaded to mitigate queuing delays.) Google-F1 and Facebook-TAO have 1M keys, with the popular keys randomly distributed to balance load. We run 3 trials for each test and 60 seconds for each trial. Experiments are CPU-bound (i.e., handling network interrupts).

## 6.2 Result Overview

NCC outperforms strictly serializable protocols dOCC, d2PL, and TR (Janus-CC) by 80%–20× higher throughput and 2–10× lower latency under various workloads (Figure 7) and write fractions (Figure 8a). NCC outperforms and closely matches serializable systems, TAPIR-CC and MVTO, respectively (Figure 8b). NCC recovers from client failures with minimal performance impact (Figure 8c). Please note that Figure 7 and Figure 8b have log-scale axes. Figure 6 summarizes the takeaway of performance improvements.



**Figure 7: NCC achieves much lower latency under read-dominated workloads with its specialized read-only transaction algorithm, 50% lower latency under write-intensive workload, and at least 80% higher throughput across workloads.**

### 6.3 Latency vs. Throughput Experiments

Figure 7 shows NCC’s overall performance is strictly better than the baselines, i.e., higher throughput with the same latency and lower latency with the same throughput.

**Google-F1 and Facebook-TAO.** Figure 7a shows the results under Google-F1. X-axis is the system throughput, and y-axis shows the median read latency in log scale. A horizontal line (O.P.) marks the operating point with reasonably low latency (< 10ms). At the operating point, NCC has a 2–4× higher throughput than dOCC and d2PL. We omit the results for Janus-CC to make the graph clearer as we found that Janus-CC’s performance is incomparable (consistently worse) with other baselines, because Janus-CC is designed for highly contended workloads by relying on heavy dependency tracking, which is more costly under low contention.

NCC has better performance because Google-F1 and Facebook-TAO have many naturally consistent transactions due to the prevalence of reads. NCC enables low overhead by leveraging natural consistency. In particular, its read-only transaction protocol executes the dominating reads with the minimum costs (Figure 6). For instance, at the operating point, NCC has about 99% of the transactions that passed their safeguard check and finished in one round trip. 99.1% of the transactions did not delay their responses, i.e., the real-time order dependencies were already satisfied when they arrived. That is, 99% of the transactions were finished by NCC within a single RTT without any delays. For the 1% of the transactions that did not pass the safeguard check initially, 70% of them passed the smart retry. Only 0.2% of the transactions were aborted and retried from scratch. All of them were committed eventually.

As a result, NCC can finish most transactions with one round of messages (for the read-only ones) and a latency of one RTT (for both read-only and read-write) while dOCC and d2PL-wound-wait require three rounds of messages and a latency of two RTTs (asynchronous commitment saves one RTT). NCC has much higher throughput than d2PL-no-wait due to its novel read-only protocol which requires one round of messages, while d2PL-no-wait requires two. The fewer messages of NCC translate to lower latency under medium

and high load due to lower queuing delay. d2PL-no-wait performs similar to NCC-RW because NCC-RW executes read-only transactions by following its read-write protocol. However, NCC-RW outperforms d2PL-no-wait under higher load because conflicts cause d2PL-no-wait to abort more frequently, while NCC-RW has fewer false aborts by leveraging the natural arrival order. This is more obvious in the Facebook-TAO results shown in Figure 7b, because Facebook-TAO has larger read transactions that are more likely to conflict with writes. The results of Facebook-TAO show similar takeaways.

**TPC-C.** Each experiment ran all five types of TPC-C transactions, and Figure 7c shows the latency and throughput (both in log scale) of New-Order while the throughput of the other four types is proportional. NCC and NCC-RW have ~20× higher peak throughput with ~10× lower latency compared to dOCC. dOCC and d2PL-no-wait have many false aborts when load increases due to conflicting writes. NCC and NCC-RW can execute most naturally consistent transactions with low costs, even if they conflict. For instance, NCC-RW has more than 80% of the transactions passing the safeguard check and fewer than 10% of the transactions being aborted and retried from scratch. NCC-RW has a 50% higher peak throughput than d2PL-wound-wait because NCC-RW requires only two rounds of messages, while d2PL-wound-wait requires three. NCC-RW has higher peak throughput than NCC because TPC-C has very few read-only transactions, which are also more likely to abort in NCC due to conflicting writes. Janus-CC’s performance benefits mostly come from unifying the transaction and replication layers and are less significant in a single-datacenter setting, especially after we made some TPC-C transactions multi-shot.

### 6.4 Additional Experiments

We show more experiments with Google-F1. We chose Google-F1 because it has both read-write and read-only transactions, while Facebook-TAO only has read-only transactions and non-transactional writes.

**Varying write fractions.** Figure 8a shows the throughput while increasing the write fraction. Each system is run at ~75% load according to Figure 7a. The y-axis is the through-



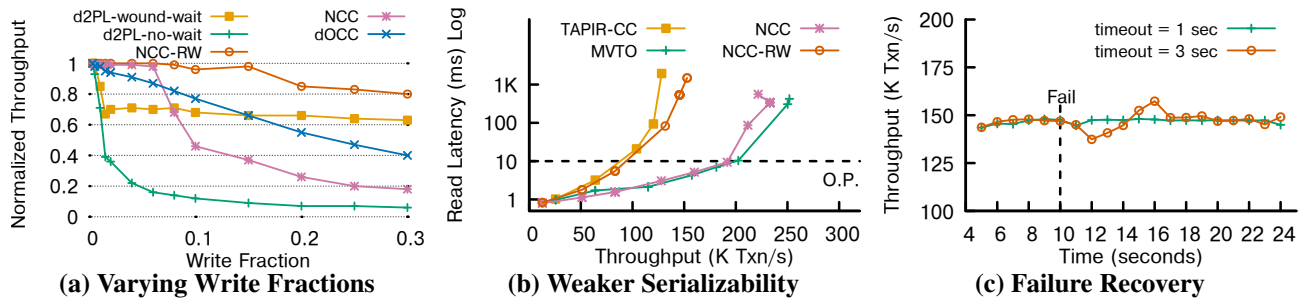


Figure 8: NCC’s performance with different write fractions (Google-WF), compared to serializable protocols (TAPIR-CC and MVTO), and under failures for the Google-F1 workload.

put normalized to the maximum throughput of each system during the experiment. The higher the write fraction, the more conflicts in the system. The results show that NCC-RW is most resilient to conflicts because NCC-RW can exploit more concurrency in those conflicting but naturally consistent transactions, i.e., NCC has fewer aborts. In contrast, other protocols may falsely abort transactions due to failed validation (dOCC) or lock unavailability (d2PL variants). NCC’s read-only transactions are more likely to abort when writes increase because frequent writes cause the client to have stale knowledge of the most recently executed writes on each server; as a result, NCC must abort the reads to avoid timestamp inversion.

**Comparing with serializable systems.** Figure 8b compares NCC with MVTO and TAPIR-CC, which provide serializability, under Google-F1. NCC outperforms TAPIR-CC because NCC has fewer messages with its read-only transaction protocol. MVTO and NCC have similar performance under low and medium load because they have the same number of messages and RTTs. Under high load, MVTO outperforms NCC when many read-only transactions in NCC are aborted: MVTO never aborts reads because it is allowed to read stale versions, whereas NCC must read the most recent version and handle timestamp inversion. In this sense, MVTO presents a performance upper bound for strictly serializable systems, and NCC closely matches the upper bound.

**Failure recovery.** Figure 8c shows how well NCC-RW handles client failures under Google-F1. We inject failures 10 seconds into the experiment by forcing *all* clients to stop sending the commit messages of ongoing transactions while they continue issuing new transactions. Undelivered commit messages cause servers to delay the responses of later transactions due to response timing control, until the recovery mechanism is triggered after a timeout. We show two timeout values, 1 and 3 seconds. NCC-RW recovers quickly after failures are detected, thus client failures have a limited impact on throughput. In realistic settings, failures on one or a few clients would have a negligible impact because uncommitted reads do not block other reads. Similarly, NCC is minimally impacted by client failures because its read-only transactions do not send commit messages and thus never delay later writes.

## 7 Related Work

NCC proposes a new strictly serializable distributed protocol. This section places it in the context of existing strictly serializable techniques, single-machine concurrency control, and techniques that provide weaker consistency. At a high-level, NCC provides better performance, addresses a different problem setting, and provides stronger guarantees, compared to these categories of work, respectively.

**General strictly serializable protocols.** As discussed in Section 2.3, existing general strictly serializable protocols are d2PL, dOCC, TR, or their variants, suffering extra costs when transactions are naturally consistent. For instance, Spanner’s read-write transactions [12], Sinfonia [4], and Carousel [68] are variants of d2PL that must acquire locks. FaRM [15], FaRMv2 [58], RIFL [31] are variants of dOCC that suffer extra validation costs, even if they use timestamp-based techniques to reduce validation aborts. AOCC [2] is a variant of dOCC and operates in a data-shipping environment, e.g., data can move from servers to client caches, which is different from NCC which works in a function-shipping environment, i.e., data resides only on servers. Rococo [51] and its descendant Janus [52] reorder transactions to minimize aborts. Granola [13] requires an all-to-all exchange of timestamps between servers, incurring extra messages and RTTs. Our evaluation shows that NCC outperforms these techniques for real-world workloads where natural consistency is prevalent. When transactions are not naturally consistent, however, these techniques could outperform NCC. Figure 9 summarizes performance and consistency properties of NCC and some representative distributed systems.

**Special strictly serializable techniques.** In addition to the general techniques discussed above, there are several interesting research directions that use specialized techniques to provide strict serializability. Some work utilizes a centralized sequencer to enforce strict serializability [6, 19, 33, 36, 45, 56, 62, 73]. Because all transactions must contact the sequencer before execution (e.g., Eris [33]), in addition to the extra latency, the sequencer can be a single point of failure and scalability bottleneck. Scaling out sequencers incurs extra costs, e.g., Calvin [62] requires all-to-all messages among

| System         | Consistency | Technique        | Latency (RTT) | Lock-free       | Non-blocking | False aborts      |
|----------------|-------------|------------------|---------------|-----------------|--------------|-------------------|
| NCC            | Strict Ser. | NC+TS            | 1             | Yes             | Yes          | Low               |
| Spanner [12]   | Strict Ser. | d2PL+TrueTime    | RO: 1, RW: 2  | RO: Yes, RW: No | No           | RO: None, RW: Med |
| d2PL-NoWait    | Strict Ser. | d2PL             | 1             | No              | No           | High              |
| AOCC [2]       | Strict Ser. | dOCC             | 2             | Yes             | No           | High              |
| Janus [52]     | Strict Ser. | TR               | 2             | Yes             | No           | None              |
| dOCC           | Strict Ser. | dOCC             | 2             | No              | No           | High              |
| d2PL-WoundWait | Strict Ser. | d2PL             | 2             | No              | No           | Med               |
| FaRMv2 [58]    | Strict Ser. | dOCC             | 2             | No              | No           | Med               |
| TAPIR [72]     | Ser.        | dOCC+TS          | 1             | Yes             | No           | Med               |
| DrTM [66]      | Ser.        | RO: TS, RW: d2PL | RO: 2, RW: 3  | RO: Yes, RW: No | No           | Med               |
| TO [7]         | Ser.        | TS               | 1             | Yes             | No           | Med               |
| MVTO [55]      | Ser.        | TS               | 1             | Yes             | No           | Low               |

**Figure 9: The consistency and best-case performance of representative distributed protocols for naturally consistent workloads, processing one-shot transactions with possible optimizations considered. NC means natural consistency, and TS means timestamp-based technique. NCC has the lowest performance costs while providing strict serializability.**

sequencers for each transaction (epoch). Some ensure strict serializability by moving all data a transaction accesses to the same machine, e.g., LEAP [35]. Some rely on program analysis and are application-dependent, e.g., the homeostasis protocol [57]. Some rely on extensive gossip messages for liveness, which lower throughput and increase latency, e.g., Ocean Vista [18] whose latency of a transaction cannot be lower than the gossiping delay of the slowest server even if this server is not accessed by the transaction. General techniques such as NCC do not have the above limitations.

**Strictly serializable read-only transaction protocols.** To the best of our knowledge, the only existing strictly serializable read-only transaction protocol that has optimal *best-case* performance is Spanner [12]. Spanner ensures strict serializability by using d2PL for read-write transactions and by using synchronized clocks (TrueTime) for read-only transactions. TrueTime must be accurately bounded for correctness and those bounds need to be small to achieve good performance, which are achieved by Google’s infrastructure using special hardware, e.g., GPS and atomic clocks [9] that are not generally available. For instance, CockroachDB [61], which began as an external Spanner clone, chose not to support strict serializability because it does not have access to such infrastructure [25]. In contrast, NCC’s read-only transactions achieve optimal best-case performance and provide strict serializability, without requiring synchronized clocks.

**Single-machine concurrency control.** Concurrency control for single-machine databases is different from the distributed setting on which this paper focuses. First, some techniques are not feasible in a distributed setting. For instance, Silo [64] relies on atomic instructions, and MVTL [3] relies on shared lock state, which are challenging across machines. Second, most techniques, e.g., Silo [64] and TicToc [69], follow a multi-phase design and would be expensive if made distributed, e.g., they need distributed lock management and one round of inter-machine messages for each phase, which would

be unnecessary costs for naturally consistent transactions. Their designs, however, are feasible and highly performant for the single-machine setting they target.

**Protocols for weaker consistency.** Many systems trade strong consistency for better performance. For instance, some settle for restricted transaction APIs, e.g., read-only and/or write-only transactions [16, 37, 38]. Some choose to support weaker consistency models, e.g., causal consistency and serializability [17, 32, 38, 39, 46, 61, 65, 70]. In contrast, NCC provides stronger consistency and supports general transactions, greatly simplifying application development.

## 8 Conclusion

Strictly serializable datastores are advocated by recent work because they greatly simplify application development. This paper presents NCC, a new design that provides strict serializability with minimal overhead by leveraging natural consistency in datacenter workloads. NCC identifies and overcomes timestamp inversion, a fundamental correctness pitfall in timestamp-based concurrency control techniques. NCC significantly outperforms existing strictly serializable techniques and closely matches the performance of serializable systems.

**Acknowledgments.** We gratefully thank our shepherd, Atul Adya, for his invaluable feedback that significantly improved this work. We thank the anonymous reviewers for their careful reading of our paper and their many insightful comments and suggestions. This work was supported by the National Science Foundation under grant CNS 1824130, 2130590, 2238768 as well as a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

**Availability.** Code and experimental scripts are available at <https://github.com/nyu-news/janus/tree/ncc>. More details on NCC are in the technical report [41].

## References

- [1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.
- [2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.
- [3] Marcos K Aguilera, Tudor David, Rachid Guerraoui, and Junxiong Wang. Locking timestamps versus locking objects. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.
- [4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.
- [5] InfiniBand Trade Association. Infiniband architecture specification, release 1.0, october 2000. <http://www.infinibandta.org/>, 2000.
- [6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
- [7] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.
- [8] Google Cloud Blog. Why you should pick strong consistency, whenever possible. <https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible>, 2018.
- [9] Eric Brewer. Spanner, TrueTime and the CAP theorem. Technical report, Google Research, 2017.
- [10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook’s distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, Jun 2013.
- [11] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems (TOCS)*, 35(1):1–37, 2017.
- [12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman and Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [13] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.
- [14] DPDK. DPDK. <http://dpdk.org/>, 2020.
- [15] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.
- [16] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.
- [17] Amazon DynamoDB. Amazon DynamoDB :: Fast and flexible NoSQL database service for any scale. <http://aws.amazon.com/dynamodb/>, 2021.
- [18] Hua Fan and Wojciech Golab. Ocean Vista: Gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1471–1484, 2019.
- [19] FaunaDB. FaunaDB :: The data API for your client-serverless applications. <https://fauna.com/>, 2021.
- [20] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.
- [21] David K. Gifford. *Information storage in a decentralized computer system*. PhD thesis, Stanford University, Department of Electrical Engineering, 1981.
- [22] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don’t matter when you can

- JUMP them! In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
- [23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [24] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1496–1499, 2008.
- [25] Spencer Kimball and Irfan Sharif. Living without atomic clocks. <https://www.cockroachlabs.com/blog/living-without-atomic-clocks/>, 2021.
- [26] Kishori M Konwar, Wyatt Lloyd, Haonan Lu, and Nancy Lynch. SNOW revisited: Understanding when ideal read transactions are possible. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.
- [27] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2013.
- [28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7), 1978.
- [29] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
- [30] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.
- [31] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushitay, and John Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.
- [32] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.
- [33] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating System Principles (SOSP)*, 2017.
- [34] Kai Li and Jeffrey F Naughton. Multiprocessor main memory transaction processing. In *Proceedings International Symposium on Databases in Parallel and Distributed Systems*, pages 177–178. IEEE Computer Society, 1988.
- [35] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2PC transaction management in distributed database systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2016.
- [36] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. MgCrab: Transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment (PVLDB)*, 12(5):597–610, 2019.
- [37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.
- [38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [39] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE International Conference on Data Engineering (ICDE)*, 2012.
- [40] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [41] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall (extended version). <https://arxiv.org/abs/2305.14270>, 2023.
- [42] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [43] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-optimal read-only transactions (extended version). Technical report, TR-005-20 v1, Princeton University, Department of Computer Science, 2020.



- [44] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.
- [45] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):2047–2060, 2020.
- [46] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment (PVLDB)*, 7(5):329–340, 2014.
- [47] Microsoft. Microsoft Azure :: New challenges need agile solutions. Invent with purpose. <https://azure.microsoft.com/en-us/>, 2020.
- [48] David Mills. *RFC1305: Network Time Protocol (Version 3) Specification, Implementation*. RFC Editor, 1992.
- [49] Radhika Mittal, Vinh The Lam, Nandita Dukkhipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.
- [50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.
- [51] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [52] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [53] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.
- [54] Costin Raiciu and Gianni Antichi. NDP: Rethinking datacenter networks and stacks two years after. *ACM SIGCOMM Computer Communication Review*, 49(5):112–114, 2019.
- [55] David P Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1):3–23, 1983.
- [56] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1747–1761, 2019.
- [57] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM International Conference on Management of Data (SIGMOD)*, 2015.
- [58] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *ACM International Conference on Management of Data (SIGMOD)*, 2019.
- [59] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.
- [60] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It’s time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. Association for Computing Machinery and Morgan & Claypool, 2018.
- [61] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.
- [62] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.
- [63] TPC. TPC-C: An on-line transaction processing benchmark. <http://www.tpc.org/tpcc/>, 2020.

- [64] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.
- [65] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.
- [66] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.
- [67] Arthur Whitney, Dennis Shasha, and Stevan Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++, 1997.
- [68] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *ACM International Conference on Management of Data (SIGMOD)*, 2018.
- [69] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *ACM International Conference on Management of Data (SIGMOD)*, 2016.
- [70] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proceedings of the VLDB Endowment (PVLDB)*, 11(10):1289–1302, 2018.
- [71] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.
- [72] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.
- [73] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *ACM International Conference on Management of Data (SIGMOD)*, 2021.



# Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta

Boris Grubic<sup>1</sup>, Yang Wang<sup>1,2</sup>, Tyler Petrochko<sup>1</sup>, Ran Yaniv<sup>1</sup>, Brad Jones<sup>1</sup>, David Callies<sup>1</sup>, Matt Clarke-Lauer<sup>1</sup>, Dan Kelley<sup>1</sup>, Soteris Demetriou<sup>1,3</sup>, Kenny Yu<sup>1</sup>, and Chunqiang Tang<sup>1</sup>

<sup>1</sup>Meta Platforms, <sup>2</sup>The Ohio State University, <sup>3</sup>Imperial College London

## Abstract

We present *Conveyor*, Meta’s software deployment tool, along with the valuable data obtained from managing over 30,000 deployment pipelines that deploy all kinds of services at Meta across millions of machines. We describe a wide range of deployment scenarios that Conveyor supports to achieve universal coverage. At Meta, out of all the deployment pipelines for services deployed via containers, 97% of them employ fully automated deployments without manual intervention: 55% utilize continuous deployment, instantly deploying every code change to production after passing automated tests, and the remaining 42% are automatically deployed on a fixed schedule (mostly daily or weekly) without manual validation. We highlight several distinguishing features of Conveyor, including safe in-place updates to reduce hardware costs, analysis of code dependencies to prevent faulty releases, and the capability to deploy complex ML models at scale.

## 1 Introduction

“*Release early, release often*” [1, 32] is central to Meta’s engineering culture. For example, Meta’s largest service, *Front-FaaS*, which is a serverless function-as-a-service platform, runs on more than half a million machines and has tens of thousands of developers making changes to its code base, with thousands of code commits every workday. Despite this extremely dynamic environment, it continuously releases a new version into production every three hours [33].

Although the concept of frequent software releases is well-established, previous studies have primarily relied on limited surveys or analyses [7, 20, 23, 25, 38–41, 48]. In contrast, we leverage our nine years of direct experience in developing Meta’s deployment tool called *Conveyor* and the wealth of data obtained from managing over 30,000 deployment pipelines to answer the following questions: 1) What is the adoption rate of deployment automation, and what is important in driving the adoption? 2) What is special about deployment safety at hyperscale? 3) What distinguishes the deployment of ML models from traditional service executables? We summarize our answers to these questions below.

## 1.1 Adoption of Deployment Automation

**Universal adoption.** We strongly argue for universal adoption of a *single* deployment tool within an organization to support all kinds of services, both small and large. At Meta, 0.1% and 1% of the largest services consume 40% and 80% of the total fleet capacity, respectively. Similarly, Google reported that “*the top 1% of jobs consume over 99% of all resources* [47].” These largest services often require the most complex deployment features, and neglecting them would lead to fragmentation and difficulties in managing the site. For example, due to FrontFaaS’s demanding requirements, it used to have its own complex deployment tool written in over 30,000 lines of code. This kind of fragmentation complicates the operation of our site. In the event of a site outage, very few people know how to safely revert a specific service’s problematic release.

Furthermore, the impact of a deployment tool on site reliability necessitates many advanced features to ensure the safety of deployments, as outlined in §1.2. While it may be tempting to develop a new custom tool to address specific needs that are currently unsupported by the standard tool, Meta’s experience has consistently shown that these custom tools, owned by individual product teams, have seldom reached the level of maturity required to provide the essential, advanced deployment-safety features. Consequently, without any exceptions, these custom tools have always been assimilated back into the standard tool as it evolves and matures.

Over the past nine years, Conveyor has achieved universal adoption at Meta, and the key to its success lies in its ability to support a wide range of deployment scenarios while ensuring the safety of deployments (§3).

**Fully automated deployments.** After addressing numerous challenges along the way, the adoption rate of *continuous deployment* [39] at Meta has greatly exceeded our initial expectations. With continuous deployment, every time a code change is committed to the code repository, it automatically goes through a series of tests. If it passes those tests, it is deployed to production immediately, without manual intervention. Currently, out of all Meta’s deployment pipelines for services deployed via containers, 97% employ fully auto-



mated deployments: 55% utilize continuous deployment and the remaining 42% automatically deploy on a fixed schedule (mostly daily or weekly) without manual validation.

In contrast, in early 2018, 47% of services at Meta were deployed manually without using any automation tool, 41% utilized an automation tool but still required manual validation, and only 12% were deployed through full automation. The significant increase in fully automated deployments, from 12% to 97%, has greatly reduced developer toil and improved productivity. Moreover, in early 2017, 20% of our fleet's RPC traffic [37] were generated by executables that had not been updated within 30 days. Currently, this number has dropped to around 1%. Up-to-date code brings many benefits, such as faster iteration speed and the timely implementation of bug fixes and security fixes.

To automate deployments, bugs and deployment failures must be embraced as a norm. Instead of introducing manual validations to prevent failures, automated guardrails such as testing and health checks should be implemented to detect release failures early and contain their adverse effects. Specifically, although 5.4% of our deployments fail, deployments are still highly reliable as the majority of those failed deployments are caught during early deployment phases with little to no production impact. The high reliability of automated deployments is evidenced by the fact that only 0.92% of finished deployments are manually reverted or patched by developers.

## 1.2 Deployment Safety at Hyperscale

**Making in-place updates safe.** Due to its strong safety guarantees, the approach of *mirroring update*, which keeps the existing deployment intact and uses a separate set of containers to deploy a new version of the software, is widely used in the industry [2, 11], as it can easily redirect traffic back to the old deployment if the new deployment encounters issues. However, the cost of keeping spare hardware for a second deployment is prohibitive for our hyperscale services. At Meta, we exclusively utilize *in-place updates* for all services, which directly update containers in the existing deployment, eliminating the need for a separate deployment.

Updating a service in place requires precise controls over container updates and health checks to ensure safety. To enable such controls, we have enhanced our cluster manager [45] to allow updating a specific subset of a job's containers to a new version while keeping the remaining containers on the old version, as opposed to the traditional approach that requires updating a job's all containers as a whole [22].

Furthermore, for complex services like sharded databases, it is essential for the service itself, rather than the cluster manager, to determine when to update each container, because the service knows best about its own requirements such as shard replica safety. Our cluster manager's TaskControl interface enables this, whereas existing cluster managers disallow it.

Finally, we have enhanced our monitoring system to conduct health checks on "*moving targets*," i.e., a dynamically

changing subset of a job's tasks. This subset evolves as the deployment progresses, in contrast to traditional health checks that always target a fixed set of tasks, i.e., all tasks in the job. Overall, these precise-control features enable Conveyor to perform in-place updates safely while eliminating the extra hardware costs associated with the mirroring approach.

**Handling complex code dependencies.** Pioneered by hyperscalers such as Google, monorepo [9], which stores the code for an organization's many projects in a single repository, has become increasingly popular due to benefits such as improved code reuse. However, increased code reuse leads to more complex code dependencies, such as a service *X* transitively depending on shared code at a depth of over 10 layers. In such cases, when a bug is introduced to some dependent code, the owner of service *X* might not even know that service *X* is affected. Our data show that about 14% of the to-be-deployed executables are affected by known bugs in dependent code and should not be deployed into production. Conveyor's Bad Package Detector automatically enforces this (§3.2), but existing deployment tools do not support it.

## 1.3 ML Model Deployment

Traditional deployment tools [3, 18, 42, 46] exclusively focus on the deployment of service executables. Even if they evolve to achieve universal coverage for service executables, in the era of rapid proliferation of ML applications, they still leave a significant gap by not addressing the deployment of ML models. Conveyor has been specifically enhanced to address this need and currently about 44% of its pipelines are for model deployments. To support ML models, Conveyor coordinates deployment pipelines for models that share the same inference executable, synchronizes the deployment of different shards of a partitioned large model, and implements phased in-place updates of models through the configuration management system [44], in contrast to the traditional approach of updating executables via the cluster manager. Dedicated ML platforms such as AWS SageMaker can deploy models using the mirroring approach [35, 36], but they do not support the advanced model-deployment features mentioned above.

**Contributions.** We make several contributions in this paper.

- We believe this paper is the most comprehensive report to date on deployment scenarios, operational experience, and production data related to software deployment.
- We demonstrate the feasibility of achieving aggressive goals for software deployment, such as frequent and automated deployment without manual validation, and using a single deployment tool to provide universal coverage for datacenter services, ML models, application configurations, host-level daemons, and mobile apps.
- We present novel techniques that ensure the safety of in-place updates, prevent faulty releases through analysis of code dependencies, and safely deploy ML models.

## 2 Overview of Software Deployment at Meta

To provide context for the discussions in later sections, this section gives an overview of Meta’s software deployment culture and deployment ecosystem.

### 2.1 Deployment Culture

At Meta, we mandate frequent software deployments for multiple reasons. First, frequent releases enhance developer productivity. Engineers at Meta heavily rely on A/B test results of new product features to guide their development. This necessitates frequently deploying and testing new code with real users. Second, frequent releases reduce the complexity of troubleshooting in production because each release contains fewer code changes. Finally, frequent releases ensure timely deployment of bug fixes and security fixes. Consider a widely publicized site outage in 2014 [29]. Two months before the outage, the issue that later caused the outage was identified and the bug fix was already committed to the code repository. Unfortunately, no new deployment took place for two months. In general, preventing human mistakes in software deployments at the scale of thousands of engineers is unachievable without utilizing automation tools like Conveyor.

Meta’s Push4Push program enforces regular deployments through tickets. Service owners get a ticket if their services are not updated within certain days (42 days for low-traffic services and 30 days for high-traffic services), and it escalates to managers at 63 days. In practice, Push4Push results in 96% of services deploying weekly or more frequently.

### 2.2 Deployment Ecosystem

Meta’s private cloud comprises multiple datacenter regions. Each region has its own instance of cluster manager called Twine [45], which manages machines and containers. During a deployment, Conveyor instructs Twine to update containers. Meta’s datacenter software is structured as many microservices [21]. A service comprises one or multiple *jobs*. A job comprises one or more *tasks*, and a task is mapped to a Linux container. Typically, a service is deployed to multiple regions for resilience, running one job for each region where it is deployed, and each of those jobs is managed by a different Twine instance.

Figure 1 presents an overview of the software deployment ecosystem at Meta. Developers define the update procedure for their services by specifying a deployment *pipeline*, which is a directed acyclic graph (DAG) comprising a set of input *artifacts* and a set of *actions*. A simple pipeline is shown at the top of Figure 1, while a more complex example is provided in Figure 3. An *action* represents an operation to be executed and takes a set of *artifacts* as input and potentially generates a new set of artifacts. Examples of artifacts include source code and compiled executables. Once all the input artifacts of a pipeline are ready, the pipeline will be executed, creating a *release*. A release is one execution of a pipeline.

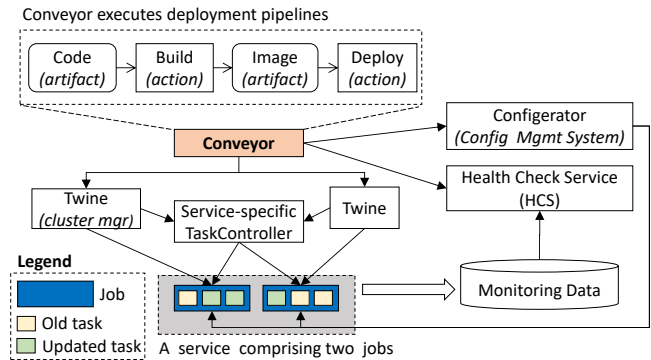


Figure 1: Software deployment ecosystem at Meta. The two Twine instances manage jobs in different datacenter regions.

The `deploy` action drives Twine to update containers and is typically the most complex part of a pipeline. To prevent a bug from instantly impacting all tasks within a job, the `deploy` action updates tasks in *phases*. Each phase updates a subset of tasks, checks their health, and proceeds to the next phase only if no issues are detected. To check service health, Twine and the service itself can log various health signals, such as CPU utilization and user engagement metrics. The Health Check Service (HCS) is responsible for checking these health signals for anomalies based on user-defined rules. For example, it can detect if user engagement drops below a certain threshold. Collecting health signals often requires a waiting period to gather monitoring data, known as the *bake time*. Therefore, a `deploy` action typically includes several phases, each with an update period and a bake period, and such information is defined in a *deployment plan*.

In addition to driving Twine to deploy service executables, Conveyor can also drive Configurator [44], our configuration management system, to implement phased deployments of ML models and configuration files (§3.3). Moreover, a service with special requirements can optionally provide its own TaskController to advise Twine on which tasks are safe to update together, as explained in the example below.

### 2.3 Component Interaction by Example

We illustrate the interaction between various components in Figure 1 through the example of deploying a new software version for a sharded key-value store (KVStore). The KVStore is deployed across two regions, denoted as  $X$  and  $Y$ , with each region running a separate job consisting of six tasks. These jobs and tasks are labeled as  $JobX = [X1, \dots, X6]$  and  $JobY = [Y1, \dots, Y6]$ . These 12 tasks collectively host 500 data shards, each of which has three replicas that are potentially distributed across regions.

According to the KVStore’s quorum protocol, if a shard loses two out of its three replicas, it becomes unavailable. Thus, concurrently updating any two specific tasks carries the risk of rendering certain shards unavailable. Since Conveyor and Twine are unaware of the application-level shard

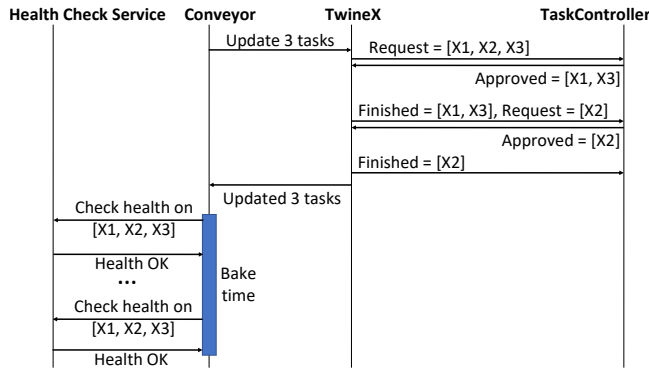


Figure 2: Phase 1 of updating the KVStore.

placement [24], they cannot determine whether it is safe to update any two specific tasks together. To ensure shard availability, the KVStore provides its own custom TaskController to advise Twine on which tasks are safe to update together.

The `deploy` action of the KVStore consists of two phases: 1) updating three tasks of *JobX*, and 2) updating the remaining three tasks of *JobX* and all six tasks of *JobY* (see §6.5.2 for alternative setups). In phase 1, Conveyor instructs *TwineX*, the Twine instance in region *X*, to update three tasks of *JobX*, as depicted in Figure 2. *TwineX* chooses to update tasks *X1*, *X2*, and *X3* and communicates this intent to the TaskController by sending `request = [X1, X2, X3]`. The TaskController responds with a subset of tasks that can be safely updated together. For example, the TaskController may find that the shard replicas hosted by *X1* and *X3* do not overlap but further including *X2* would result in some shard losing more than one replica. Therefore, it responds with `approved = [X1, X3]`. Consequently, *TwineX* executes the updates for *X1* and *X3* and then notifies the TaskController. The TaskController subsequently responds with the next batch of tasks that are now safe to be updated, in this example, `approved = [X2]`.

Once all tasks in phase 1 of the `deploy` action are updated, Twine notifies Conveyor. Next, during the configured `bake` time, Conveyor periodically invokes the Health Check Service to access the health of *X1*, *X2*, and *X3* (instead of all tasks in *JobX*), by comparing the error rate, latency, and user engagement metric of those tasks before and after the update.

If the health checks pass, Conveyor proceeds to phase 2 of the `deploy` action by instructing *TwineX* to update the remaining 3 tasks of *JobX* and instructing *TwineY* to update all 6 tasks of *JobY*. The process is similar to that in phase 1.

This example highlights a key principle in our design: separation of concerns. Conveyor orchestrates the execution of the deployment plan supplied by the service owner without worrying about how tasks are updated. A custom TaskController usually has a simple implementation since it only needs to determine which tasks can be safely updated together based on the application constraints. The complexity of actually updating tasks and managing their lifecycle is handled by Twine without the involvement of Conveyor or TaskController.

## 3 Deployment Scenarios and Solutions

To achieve universal coverage, Conveyor supports a wide range of deployment scenarios. This section presents these scenarios and the corresponding solutions in Conveyor.

### 3.1 Enabling In-place Updates

The software deployment approach affects hardware costs. The in-place update approach restarts an existing task on the same machine to run the new executable. In contrast, the mirroring approach, which is also called Red-Black [43] or Blue-Green [2] deployment, first starts a new job with the new executable, often on other machines, gradually redirects the traffic from the old job to the new job, and finally shuts down the old job. Although this approach is safer as the traffic can be quickly redirected back to the old job if the update fails, it needs extra hardware to run both the old and new jobs in parallel. Consider the example of deploying Front-FaaS to 500K machines every three hours. A naive mirroring approach would require 500K extra machines, which is unacceptable. One optimization is to divide it into many small jobs and utilize mirroring to update one small job at a time. However, this approach would result in the loss of quick rollback capability and complicate job autoscaling [16, 34]. Although further optimizations might be possible, the resulting solution would not necessarily be cheaper, simpler, or more generic than in-place updates.

Despite the benefits of hardware savings, the in-place update approach lacks widespread support from existing deployment tools due to the difficulty of ensuring deployment safety. Below, we present how we have made in-place updates safe and practical by co-designing our deployment tool (Conveyor) and our cluster manager (Twine [45]).

**Collaborative control between Conveyor and Twine.** As discussed in §2.2, during each phase of the `deploy` action, Conveyor instructs Twine to update a specific number or percentage of tasks, denoted as  $N_{big}$  and then waits for a period of time to collect comprehensive health signals before moving on to the next phase. Twine, however, does not update  $N_{big}$  tasks all at once. Instead, it updates only  $N_{small}$  tasks in one batch, where  $N_{small}$  is much smaller than  $N_{big}$ , to avoid losing too many tasks simultaneously. Twine then checks the liveness of each task before proceeding to update the next  $N_{small}$  tasks. Similar to other cluster managers [22], Twine’s liveness check is rudimentary and only verifies that an individual task is running properly. However, it is unable to detect subtle issues such as the new code’s memory regression compared to the old code, which is handled by Conveyor’s comprehensive health checks (§3.2).

The collaborative control between Conveyor and Twine enables fast and safe deployments by assigning the most suitable functions to the right layers. Let’s consider some alternative designs. If Conveyor instructs Twine to update  $N_{small}$  tasks instead of  $N_{big}$  tasks, and then waits for a period of time to collect comprehensive health signals, the deployment speed



would be too slow. On the other hand, if Twine updates  $N_{big}$  tasks instead of  $N_{small}$  tasks in one batch, the service might lose too much capacity and become overloaded. Finally, eliminating Twine’s rudimentary task liveness check would mean a destructive bug could affect  $N_{big}$  tasks instead of just  $N_{small}$  tasks before being detected.

Unlike the close collaboration between Conveyor and Twine, in the widely used open-source setup of Spinnaker [42] (deployment tool) instructing Kubernetes [22] (cluster manager) to update containers, Spinnaker cannot control the size of each phase,  $N_{big}$ , which defaults to the size of the entire job. This is because Kubernetes disallows partial-job updates. This simplistic approach is not suitable for in-place updates, because, when the comprehensive health checks detect a bug, it is likely that all tasks of the job have already been updated.

**Pluggable TaskControl.** In the task-update protocol described above, most services can use a per-service constant  $N_{small}$  and have no constraints on the specific tasks to be updated. However, the sharded key-value store described in §2.3 provides an example of services that require more precise control of  $N_{small}$  and the specific tasks to be updated. Figure 2 further illustrates how such a service can implement a custom TaskController to ensure safe in-place updates. TaskControl, in general, enables services to have precise control over task updates for various reasons, not limited to data shard availability. For example, FrontFaaS utilizes TaskControl to maximize its deployment speed (§4). Further details on TaskControl can be found in our previous work [24, 45].

**Hardware failure and planned maintenance.** When multiple tasks undergo in-place updates simultaneously, there is a risk of reducing the available capacity of a service to an unhealthy level. Merely controlling the update speed through the deployment pipeline is insufficient since certain tasks may be in an unhealthy state due to machine failures or planned maintenance, which Conveyor is unaware of. To tackle this issue, the service owner can inform Twine of a budget that indicates the maximum number or percentage of tasks that can be offline for any reason, such as task update, hardware failure, or planned maintenance. If the budget is projected to be exceeded, Twine pauses task updates.

**Zero downtime hotswap.** Our routing service in edge datacenters forwards user-facing traffic to our datacenters and holds live HTTPS connections to user devices. Naively restarting a task for an update would cause user-facing errors. Twine provides a same-host hotswap feature to solve this problem. It first starts a new task on the same machine, which binds to the same TCP ports as the old task. Then the new task and the old task cooperate with each other to hand over the live connections from the old task to the new task with the help of eBPF [14]. One limitation of hotswap is that it requires the container to be configured with sufficient memory to start two tasks, which is the reason why it is not used universally.

**Summary.** All of the aforementioned deployment scenarios with in-place updates cannot be properly implemented

without support from the cluster manager. We believe this is a key reason why existing deployment tools primarily use the mirroring approach, since the cluster managers they rely on do not provide the necessary functions for in-place updates.

## 3.2 Deployment Safety

To enable continuous deployment, we accept bugs and deployment failures as a norm and rely on automated guardrails such as testing and health checks to detect release failures early and mitigate their adverse effects. In this section, we describe techniques that help Conveyor deploy safely.

**Moving-target health checks.** After each deployment phase, Conveyor invokes the Health Check Service (HCS) to evaluate the health of the service. Multiple health checks can be associated with a job, and each health check specifies a data source such as a time series database for monitoring data [31], a metric, data transformations (e.g., calculating specific percentiles), and a decision threshold. The metrics encompass system metrics (CPU, memory, crashes), RPC metrics (connections, errors), and application-level business metrics. The thresholds can be absolute (e.g., fail if CPU utilization exceeds 90%), A/B comparative (e.g., fail if the new task’s CPU utilization exceeds that of the task that still has not been updated by 10%), or time-based (e.g., fail if the CPU utilization increases by 10% since the deployment started). By default, a failed health check triggers Conveyor to revert the release.

In contrast to traditional monitoring systems that track the health of an entire job, in-place updates require the HCS to track “*moving targets*”, i.e., a dynamic subset of tasks that change throughout different deployment phases. Achieving this level of precision and adaptability necessitates a seamless integration between Conveyor, Twine, and HCS. Specifically, Twine assigns unique identifiers to tasks, enabling differentiation between the old and new tasks. The monitoring data for tasks is tagged with these identifiers. Depending on the current deployment phase, Conveyor dynamically instructs HCS to perform health checks on tasks with specific identifiers.

**Code dependency analysis to prevent faulty releases.** A monorepo [9] stores the code for an organization’s many projects in a single repository, promoting code reuse but also leading to increased code dependencies. For instance, the ServiceRouter [37] library is compiled into nearly every service in Meta, and it may rely on a high-performance data structure library, which in turn may rely on a profiling library, and so on. In a monorepo setup, whenever a new version of a library is committed, any services that depend on the library will be automatically compiled with the new version. Consequently, the owner of a service may not even be aware that their service is affected by a bug in a shared library. To tackle this issue, Conveyor offers the Bad Package Detector (BPD). If library developers discover a bug in the library, they can report it to Conveyor. The BPD then utilizes a code dependency graph, which is provided by our build system [10], to identify and



cancel the releases of all service executables that were built with the problematic version of the library.

Accurate code dependency analysis poses a challenge for the BPD as it requires finding the right balance between false negatives and false positives. Achieving perfect coverage would entail considering all possible direct and indirect dependencies of a service, which is often impractical. To strike a balance, the BPD currently tracks 14 levels of dependency. Our production data reveals that about 14% of to-be-deployed executables are invalidated by the BPD. This highlights the importance of handling bugs in dependent code.

**Comprehensive testing.** Conveyor supports various types of tests in deployment pipelines. The `PerfTest` tool records and replays production traffic to perform A/B testing between old and new code, while the `IntegrTest` [28] tool sets up interdependent services and tests their interactions. Additionally, `IntegrTest` can perform randomized fuzzing tests. `Canary` updates a small number of production tasks with new code and collects health signals, enabling direct testing in production. Multiple `canaries` can be executed in parallel to test different code variations. Even if a release’s `deploy` action is not chosen for final execution (e.g., a release’s `deploy` action, while waiting to start on Monday at 9 AM, gets superseded by a newer release), it is still valuable to execute the test actions to detect bugs as early as possible.

**Summary.** While other deployment tools also support testing and health checks, they lack some key capabilities. In contrast to HCS’ ability to track “*moving targets*,” traditional health checks rely on alarms defined for an entire job, which is insufficient to support in-place updates. Moreover, bug dependency analysis is not supported by existing deployment tools. Finally, we are not aware of any other large-scale adoption of record and replay as a generic platform for performance tests.

### 3.3 ML Model Deployment

Deployments of ML models for inference have emerged as an important issue, given the rapidly increasing number of ML applications. While existing deployment tools generally do not handle model deployments, Conveyor has been specifically enhanced to address this need and currently about 44% of its pipelines are for model deployments. Below, we describe Conveyor’s support for model deployments.

**Deployment via configuration change.** In Conveyor’s first implementation for model deployments, model update and executable update share the same pipeline, requiring Twine to restart the container. However, as model updates may occur more frequently than updates to inference executables, frequent container restart results in a frequent loss of expensive GPU capacity for request serving. Moreover, since a model often contains gigabytes (GBs) of data, the time required to load GBs of data during the restart can be lengthy.

To solve this problem, some inference services utilize two orthogonal pipelines. One pipeline deploys the inference executable through Twine, while the other deploys the model

data through Configurator [44], Meta’s configuration management system. To track model updates, all tasks serving a model subscribe to a configuration that specifies the current version of the model to be served. When a new version of the model becomes available, instead of exposing it to all tasks simultaneously, Conveyor instructs Configurator to incrementally expose the new version to tasks in phases, following the deployment pipeline. Configurator utilizes a data-distribution tree to notify the tasks in a scalable manner. Once the tasks receive the notification of a new model version, they utilize Owl [15], a peer-to-peer data-distribution system, to fetch the new model. While still serving live requests, a task merges the new model into the old model piece by piece without consuming additional memory as it never keeps full copies of both models in memory simultaneously. Overall, Conveyor ensures safe deployments of models through phased releases, which are meticulously managed via configuration changes.

Conveyor’s ability to perform phased deployments of generic configuration changes extends beyond its use in ML model updates. Conveyor pipelines are widely utilized to ensure safe deployments of various configuration changes.

**Lockstep deployment of interdependent services.** Some of our ML models are too large to fit in one machine’s memory, so they are partitioned into interdependent shards, each including multiple replicas for fault tolerance and throughput. Each shard is mapped to a different job, and typically the first shard serves as the aggregator to combine results from other shards. However, updating different shards independently may cause compatibility issues, because combining outputs from different versions of the shards will produce incorrect results. To ensure compatibility, replicas of the first shard are configured to only receive outputs from replicas of other shards of the same version. This design requires Conveyor to perform a lockstep deployment of different shards to avoid capacity loss during deployment. For instance, 5% of each shard’s replicas are updated at the same time and 5% of the client traffic is directed to the new version, before proceeding to update 10% of each shard’s replicas, and so forth.

**Parent-child pipelines.** Meta’s ML inference system serves tens of thousands of ML models using about 10 different inference executables. Each model, along with its inference executable, is deployed via a separate pipeline. Since many models share the same inference executable, updating one executable may cause thousands of pipelines to initiate a new release at the same time. This not only causes a load spike on Conveyor and Twine, but also increases the risk of an undetected bug in the inference executable impacting many models simultaneously.

While existing deployment tools manage each pipeline in isolation, Conveyor coordinates releases across pipelines that share common artifacts by setting up a parent-child relationship between them. Specifically, each inference executable is managed by a parent pipeline, which includes sophisticated testing but no `deploy` action, while the pipelines for models

served by the executable act as its child pipelines and include the `deploy` action. When updating an ML model without modifying the executable, only the corresponding child pipeline is executed, without involving the parent pipeline. However, when updating the executable, the parent pipeline is executed first. If successful, all the corresponding child pipelines are then executed with randomized delays to avoid starting them at the same time and overloading the system. Moreover, it can be configured in such a way that a subset of child pipelines for less important models is executed first to help detect issues with the executable.

### 3.4 Advanced Features for Universal Adoption

To achieve universal adoption, Conveyor must support advanced use cases. We describe them in this section.

**DAG pipelines.** Conveyor initially modeled deployment pipelines as a sequence of sequential stages, such as `build`→`test`→`deploy`. However, this pattern has not generalized to complex services like FrontFaaS, which may build, test, and deploy two different versions in parallel (§4). The sequential pipeline is overly restrictive in that it requires testing for both versions to finish before the deployment for any version can start. Therefore, we have improved Conveyor by modeling its pipelines as directed acyclic graphs (DAGs). These DAGs support conditions, branching, and mutual exclusion groups. Since multiple releases of a pipeline may be executed in parallel, Conveyor allows users to define actions as a mutual exclusion group, meaning that concurrent releases should not execute these actions in parallel. For example, if a pipeline includes a `load-test` action and a `deploy` action, they may be put in an exclusion group so that a `deploy` action in one release will not accidentally fail its health check due to a `load-test` from another concurrent release.

**Mutable artifacts.** Conveyor initially mandated one `build` action at the beginning of a pipeline, resulting in an immutable artifact, the executable, to be used throughout the pipeline. However, this simple model does not fit well with complex deployment scenarios. One example is feedback-directed optimization (FDO) [12], which involves profiling executables in production and using the profiling data to guide recompilation of the code, resulting in an updated artifact, the new executable. Therefore, Conveyor has been extended to support mutable artifacts and allow multiple `builds` within one pipeline. To perform FDO, for example, the pipeline can first build the baseline executable, deploy it to a small number of tasks that receive production traffic, and collect profiles. It then builds the executable again using FDO and finally deploys the optimized executable to all tasks.

**CLI and daemon deployment.** Every machine in our fleet runs a set of CLI tools and daemons that provide utility functions. As these host-level executables are not managed by Twine, Conveyor provides a `deploy` action type called *Slowroll* to manage them. Due to the massive scale of deploying these executables to every machine, Slowroll adopts a

pull model instead of Twine’s push model. In the pull model, each machine periodically downloads the new version of the software being deployed. These deployments can take a long time to finish. For example, due to the massive scale of deploying a specific daemon to every machine, its pipeline has 43 phases and a deployment can take more than a week. Moreover, some CLI tools are infrequently used, requiring a significantly longer “bake time” (e.g., 8 hours) to collect health signals.

### 3.5 Software Backward Compatibility

Despite backward compatibility being a generic requirement for software deployment, it is largely left for services to handle because it often involves application-specific logic.

**API backward compatibility.** During a deployment, the new and old versions of a service will coexist for a while. When the API of a service needs to be changed, a common practice at Meta is to support both the old and new APIs and gradually switch clients to invoke the new API. Once all clients are migrated to the new API, the code for the old API can be deleted. This is called *N-1* compatibility, meaning that in addition to the current version *N* of the API, it also supports version *N-1* but not version *N-2* or older.

**Database backward compatibility.** When a service update involves data migration from one database to another or an upgrade of the database schema, extra care is needed. For example, when switching from an old database to a new database, a common strategy is to perform double writes: the service writes new data to both databases but always reads data from the old database, while a background process migrates old data from the old database to the new database. This strategy simplifies the rollback process as it only requires deleting the new database and redeploying the old code.

### 3.6 Summary of Distinguishing Features

While Conveyor and Twine provide many features to achieve universal adoption, we consider that the following features distinguish Conveyor and Twine from existing tools most.

1. In-place updates allow us to minimize hardware costs, which is important while operating at hyperscale.
2. TaskControl provides flexibility for services to precisely control the speed and sequence of their task updates.
3. The ability to update a subset of tasks and perform health checks on moving targets enables Conveyor to exert fine-grained control over tasks, enabling in-place updates.
4. In a monorepo [9] setup, it is important to perform code dependency analysis to prevent faulty releases.
5. Features to support ML model deployments have grown to become a first-class citizen in Conveyor.
6. Conveyor provides a one-tool-fits-all solution for safe deployments of various artifacts, such as service executables, ML models, and application configurations.

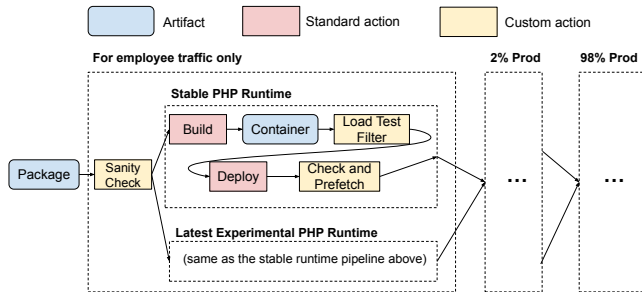


Figure 3: Simplified deployment pipeline for FrontFaaS.

Existing deployment tools lack these features. Without them, Conveyor would not have achieved universal adoption.

Conveyor is designed to be a generic and extensible deployment tool, and does not burden itself with every feature needed by every service. Its extensible architecture allows services to easily implement their own TaskControllers, actions, and new types of artifacts. Moreover, sometimes it is preferable to consider a service redesign to leverage Conveyor’s standard functions. For instance, Meta’s ML inference platform initially hosted multiple ML models as separate processes within a single container, and requested Conveyor to provide the feature of using independent pipelines to update different processes within the same container. Since this feature is complex and not needed by other services, we did not support it in Conveyor and the ML inference platform was ultimately redesigned to adopt the one-model-per-container approach.

## 4 Case Study of FrontFaaS

To illustrate the end-to-end usage of Conveyor, we present a case study of FrontFaaS, a serverless Function-as-a-Service (FaaS) platform for PHP functions. It differs from other serverless platforms such as AWS Lambda [5] in several ways: 1) it only hosts synchronous functions that clients directly invoke, while event-driven asynchronous functions are hosted by another platform; and 2) for efficiency, a single PHP runtime process can execute multiple functions concurrently. Since FrontFaaS is serverless, developers simply commit function code without worrying about code deployment or server provisioning. Each year, tens of thousands of developers commit serverless function code to FrontFaaS, with thousands of code commits each workday. Currently, FrontFaaS runs on over half a million machines and makes a new release every three hours to deploy the code of all functions together [33]. Through FrontFaaS, Meta developers create PHP functions that are servicing traffic from end users when they visit Meta web pages. Note that many Meta employees are end users of Meta products as well, and their traffic is often used for testing purposes as discussed next.

Figure 3 shows a simplified deployment pipeline for FrontFaaS. To enable phased deployments, the machines hosting FrontFaaS are partitioned into three pools: one for servicing employee traffic, one for servicing 2% of production traffic,

and one for servicing the remaining production traffic. Accordingly, the deployment pipeline is divided into three major phases, one for each machine pool. A release proceeds to the next machine pool only if the deployment to the prior pool succeeds. Each pool is split into two sub-pools: one processing a small amount of traffic with an experimental version of the PHP runtime [30] and one processing the remaining traffic with a stable version. If the experimental version outperforms the stable version, it will become the new stable version.

The first step in the deployment pipeline is a custom `sanity-check` action that performs a FrontFaaS specific logic to ensure that there are no deployment-blocking alerts. Then, a standard `build` action is used to build the container image. Independent of specific releases, load tests always continuously run on certain machines to gather performance data and build capacity models. The custom `load-test-filter` action excludes machines that are scheduled to be load-tested from being monitored by health checks, to avoid load-test induced false alarms during health checks. The pipeline then uses a standard `deploy` action to update tasks and run health checks. Finally, a custom `check-and-prefetch` action runs the sanity check again while asking Twine [45] to prefetch FrontFaaS’ container image on machines that will be updated soon. This prefetch reduces the overall duration of each release by 5-30%.

Since FrontFaaS continuously deploys every three hours across more than half a million machines, fast deployment is an important requirement. Within a `deploy` action, FrontFaaS relies on two techniques to accelerate a deployment. First, it implements a custom TaskController that controls the rate at which tasks are updated based on the CPU utilization of FrontFaaS jobs. It tries to concurrently update as many tasks as possible, as long as the temporary loss of those tasks does not cause other FrontFaaS tasks to handle too much traffic and become overloaded. During off-peak hours, updates can be applied to many tasks in large batches, while during peak hours when traffic is high, the Task Controller applies updates in smaller batches to prevent overload. Thanks to this optimization, deployments during off-peak hours are approximately three times faster than those during peak hours.

The second technique to accelerate a deployment is to update tasks and perform health checks in parallel, but the short health-check time requires FrontFaaS to have highly accurate health checks. FrontFaaS primarily relies on three health-check metrics: 1) the number of fatal errors, 2) the number of unavailable tasks, and 3) the write rate of error logs. Every minute during a `deploy` action, these metrics are averaged over the previous three minutes and compared to the average during a three-minute period before the deployment began. If any datacenter region experiences an increase in these metrics above a certain threshold, Conveyor pauses all updates in that region. If too many jobs are paused, Conveyor considers the deployment a failure and initiates a rollback.



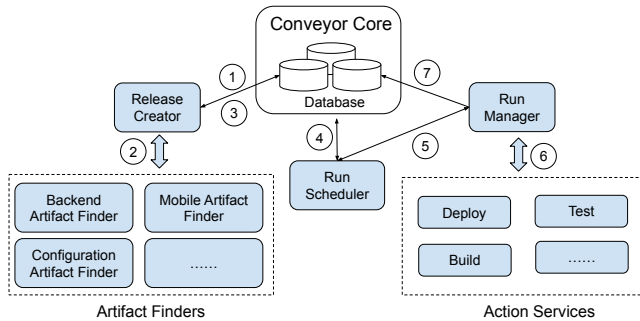


Figure 4: Conveyor Architecture.

Overall, FrontFaaS achieves fast and safe deployment at hyperscale by utilizing TaskControl and multiple custom actions made possible by Conveyor’s extensible architecture.

## 5 Design and Implementation of Conveyor

In this section, we briefly summarize Conveyor’s design and implementation.

### 5.1 Conveyor Design

Figure 4 depicts the architecture of Conveyor. Conveyor comprises several stateless services that share a database. The database stores user-defined pipelines, metadata, as well as the last step executed for each release and action. To accommodate the high throughput due to tens of thousands of pipelines, the database is partitioned into many shards.

For robustness and scalability, all components of Conveyor adhere to the worker-pool paradigm. They store the IDs of pipelines, releases, and action runs in a queue. Multiple workers then periodically scan the queue, identify the required operations, and execute them. The number of workers can be scaled up or down based on the load. The execution of a pipeline involves the following steps.

**Create release (Steps 1-3 in Figure 4).** Each pipeline accepts a set of input artifacts. The Release Creator periodically scans for new input artifacts (Step 1) by invoking Artifact Finders (Step 2). When new artifacts are discovered, Conveyor creates a “release object” in the database, thereby triggering the execution of the pipeline (Step 3). We provide seven standard Artifact Finders, and allow users to implement custom Artifact Finders. All Artifact Finders implement a single method, `getArtifacts()`, which returns a set of discovered artifacts. For example, one Artifact Finder identifies latest commits that successfully passed unit tests, while another identifies executables that have been already built and marked with a specific tag.

**Schedule (Step 4 in Figure 4).** The Run Scheduler schedules the execution of releases and actions by periodically scanning through all releases and stepping through each action in each release’s corresponding pipeline. When the input artifacts for an action exist and all preceding actions have success-

fully completed, the Run Scheduler schedules the action for execution.

Because Conveyor allows for multiple active releases derived from one deployment pipeline, the Run Scheduler must coordinate actions across these releases. Some actions, like `canary`, permit concurrent execution on multiple releases, while others, like the `deploy` action, only allow execution on a single release at a time. Therefore, the Run Scheduler must determine which active release executes the `deploy` action. Consider a case where a pipeline deploys to production on every Monday at 10 AM. If multiple code changes have occurred since the last deployment, two releases might be ready to run the `deploy` action on Monday at 10 AM. In this case, the Run Scheduler will cancel the older release.

**Run actions (Steps 5-7 in Figure 4).** The Run Scheduler determines when an action should be executed and notifies the Run Manager accordingly (Step 5). The Run Manager then initiates the execution of the action by invoking a service that implements the action’s logic (Step 6). When executing an action, the Run Manager first invokes the action’s `startRun()` method and then periodically calls its `getRunProgress()` method to track the progress. The outcome of the action is recorded back to the database (Step 7).

Conveyor offers several standard actions, including the `build` action for compiling source code and running corresponding unit tests, various testing actions, the `pkg` action for tagging executables for easy naming and access, the `deploy` action for deploying the new version to production, the `CustomScript` action for running any user-defined script (e.g., to shift traffic before a `deploy` action starts), and the `ManualPick` action for pausing a pipeline and awaiting the service owner’s decision. Conveyor’s extensible architecture allows users to create custom actions by implementing the `startRun()` and `getRunProgress()` methods.

### 5.2 Implementation of the Deploy Action

Since the `deploy` action is the most complex part of Conveyor, we describe it in more detail below. Like a sub-pipeline within the greater Conveyor pipeline, the `deploy` action’s deployment plan specifies the number of phases, which jobs and tasks to update in each phase, the baking time after each phase, and the success criteria. Conveyor supports both percentage and task-count based configuration when specifying the success criteria and the tasks to be updated.

A few widely used deployment plans are commonly employed. For small services or services that prioritize deployment speed, a common plan is to request Twine to update all of their tasks in a single phase. Therefore, the number of tasks to update in a phase,  $N_{big}$ , as described in §3.1, equals the total number of tasks in the job. However, please recall from §3.1 that Twine is still configured to update only  $N_{small}$  tasks at a time to avoid losing too many tasks simultaneously. For services that prioritize safety, their jobs are often updated region by region, as our services are always designed to toler-



ate the loss of a whole region. To balance safety and speed, a common strategy is to update an exponentially increasing number of tasks in each phase (e.g., 1% of tasks in the first phase, 10% in the second, and 100% in the last), assuming that a bug is likely to be revealed in the early phases.

For services requiring high deployment speed, Conveyor can update tasks and perform health checks in parallel. As described in §3.1, during a deployment phase, Conveyor by default requests Twine to update a group of  $N_{big}$  tasks, waits for the updates to finish, and then periodically performs health checks during the bake time. The whole process can take a long period of time. With the parallel approach, Conveyor submits the request for Twine to update  $N_{big}$  tasks and then immediately performs health checks while the update is still in progress. This approach saves the wait time but necessitates the service to have highly accurate health checks due to the short health-check time.

Finally, users have the option to deploy to environments that are not managed by Twine by implementing custom *deployment types*. For instance, we maintain a pull-based deployment type for CLI tools and Linux daemons running on bare-metal machines (§3.4). Furthermore, users have created over 20 custom deployment types for various non-standard deployment targets, such as VMs in public clouds, application configurations, and serverless stream-processing functions.

To implement a custom deployment type, a user needs to build a service that implements a few methods. When initiating a new `deploy` action, the `fetchDeployUnits()` method is invoked, returning a list of *deploy units*. Each deploy unit represents a group of tasks to be updated. Then the `executeUpdates()` method is called to update a set of deploy units with a specific set of artifacts for deployment. Finally, the `deploy` action periodically calls the `trackUpdates()` and `runHealthChecks()` methods until all updates are completed. If any health checks fail, the `deploy` action fails early.

### 5.3 Availability, Reliability, and Recoverability

Currently, Conveyor is implemented in 360K lines of Rust code, including test code and utility tools, and its components run on several hundred machines. Conveyor is not the performance bottleneck in software deployment, as other tools that Conveyor relies on, such as Twine, often perform more extensive work than Conveyor itself.

For high availability, both the database used by Conveyor and each Conveyor component are replicated across multiple datacenter regions. However, it is not necessary to replicate every Conveyor component in every region. Conceptually, one global setup of Conveyor manages the deployments of all services across all regions. A Conveyor component in region  $X$  can communicate with the Twine instance in region  $Y$  to remotely drive the deployment of services in region  $Y$ .

Presently, Conveyor's service level objective (SLO) is to ensure that less than 0.5% of deployments fail due to issues in Conveyor or any of its dependencies, such as Twine, Health

Check Service, build service, and Configurator. Conveyor consistently meets or exceeds this SLO.

The circular dependency between Conveyor and Twine poses challenges to their recoverability. Conveyor consists of a set of services that are deployed via Conveyor itself and hosted inside containers managed by Twine. Similarly, Twine is also implemented as a set of services that are deployed via Conveyor and hosted inside containers managed by Twine itself. When Conveyor or Twine fails, the entire ecosystem cannot update itself. To address this issue, we have designed them to be self-recoverable whenever possible and have introduced manual recovery tools for worst-case scenarios. In the event that Conveyor fails and cannot deploy bug fixes for itself, direct commands can be issued to Twine to start new jobs for Conveyor with the proper bug fixes.

To set up Twine to manage itself, we have implemented a two-layer deployment approach. The top layer consists of two independent instances of Twine, and under normal conditions, one instance can manage and update the other. These top-layer instances are responsible for managing and updating the numerous Twine instances in the bottom layer, which in turn manage and update user jobs. As long as at least one of the top-layer's Twine instances is functioning properly, all Twine instances can be updated normally. In the event that both top-layer Twine instances experience malfunctions, we have a dedicated tool that can be used to directly bootstrap a top-layer Twine instance and initiate the recovery process.

### 5.4 Lessons from Conveyor's Evolution

Over the course of nine years, Conveyor has undergone significant evolution, progressing from v1 to v2, and eventually to v3. Each subsequent version represents a complete system rewrite that incorporates the valuable lessons we have learned.

As a CLI tool, Conveyor v1 enables service owners to manually initiate phased in-place updates of services. Service owners could define the rollout phases and health checks. The Health Check Service (HCS) was developed along with Conveyor from the very beginning. Despite Conveyor v1 being relatively simple, about 12% of services adopted it, demonstrating a strong need for a standard deployment tool.

The biggest change from Conveyor v1 to v2 was to make it a long-running service so that it could automatically start deployments on a pre-configured schedule without manual intervention. We also introduced a more complete pipeline model, where a pipeline consisted of a sequence of phases, each comprising a set of actions. Several enhancements were implemented, including support for the Bad Package Detector (BPD), parent-child pipelines, and pull-based deployments. To boost adoption, Conveyor and Twine were made extensible by providing various interfaces for custom integrations, such as TaskControl, custom actions, and custom artifact finders. As a result of these features and the company-wide Push4Push program, about 94% of services adopted Conveyor.

The remaining 6% of services that did not adopt Conveyor were the largest and most complex ones, requiring more advanced features. Furthermore, the rapid adoption of Conveyor exposed its limitations in terms of performance and reliability. As a result, Conveyor v3 was introduced as another complete rewrite, this time switching from Python to Rust. The data model evolved from a sequential pipeline to a DAG. Additional features were implemented, including lockstep deployment, mutable artifacts to support feedback-directed optimization (FDO), deployment of mobile apps and application configurations, and better support for ML models and FrontFaaS. These enhancements helped Conveyor achieve universal adoption.

One ongoing evolution of Conveyor v3 is to enhance its real-time responsiveness. In Conveyor’s current architecture, the latest state of each release is stored in a database, and a group of Conveyor workers periodically poll the database to identify actions that are ready for execution. We chose this periodic polling design for its robustness. However, the deployment of ML models and application configurations now requires faster execution of pipelines that cannot be supported by the polling method. Therefore, in Conveyor’s new design, the completion of one action will immediately trigger the execution of the next action without any delay. However, we will still maintain the polling mechanism as a reliable fallback to safeguard against any transient failures.

## 6 Evaluation in Production

In this section, we use production data to help answer the following questions:

1. Has Conveyor achieved universal coverage?
2. Do developers trust fully automated deployments?
3. Are Conveyor’s deployment-safety mechanisms effective?
4. How often do deployments fail and why do they fail?
5. What are the observed patterns in pipeline setup, and what are the best-practice recommendations for pipeline setup?

Using three weeks of data from April to May 2023, we studied all deployment pipelines, which amounted to more than 30,000 pipelines. We divide them into four categories:

- *Regular services*: 24.4% of the pipelines deploy traditional non-ML services through containers managed by Twine [45]. This category serves as the primary point of comparison with other deployment tools, as those tools may not support the other categories listed below.
- *Large services*: 0.45% of the pipelines deploy the largest services that consume 80% of our fleet’s total capacity, with each of them using at least 7,700 servers. These large services are a subset of regular services.
- *ML models*: 44.4% of the pipelines deploy ML models, predominantly through Twine, with some utilizing Configurator [44] to control when a task consumes a new model.
- *Other pipelines*: The remaining 31.1% of pipelines are used for various purposes, such as running tests without per-

forming an actual deployment. The data we report for this category will only include those with at least one `deploy` action, which amounts to 6.7% of all pipelines. These pipelines deploy artifacts that are not managed by Twine, such as CLIs, daemons, configurations, and mobile apps.

### 6.1 Universal Coverage

It is challenging to precisely calculate the percentage of services that should utilize Conveyor but do not use it, primarily due to the presence of experimental services that will never be deployed in production. Interviewing the owners of tens of thousands of services to determine this percentage would be impractical. Instead, we focus on calculating the coverage for all 195 largest services. These services tend to be complex, making the adoption of Conveyor more challenging compared to other services.

Conveyor achieves 100% coverage for these large services, with the following caveats: 1) One of them is an ML training job, where software updates during its training run are intentionally avoided. 2) Five of them are short-lived experimental services that do not need automated deployments as they will not be deployed into production. Overall, the advanced features described in §3 enable Conveyor to achieve universal coverage, even for the most complex services.

### 6.2 Trust in Fully Automated Deployments

To understand whether developers trust fully automated deployments, we classify pipelines into five categories based on their release schedules: 1) *Continuous deployment*, which is executed immediately whenever the input artifacts are ready; 2) *Daily*, which is executed at least once every working day at a fixed time, such as 9AM; 3) *Weekly*, which is executed at least once each week; 4) *Biweekly/monthly*, which is executed biweekly or monthly; 5) *ManualPick*, which may automatically execute early actions such as small-scale deployments, but requires human confirmation before executing the final stage of large-scale `deploy` actions.

Table 1 shows that among regular services, 54.6% adopt continuous deployments, and in total 96.5% adopt fully automated deployments without manual validation. These results indicate that with the guardrails provided by testing, health checks, and automated revert of faulty releases, developers release often and trust fully automated deployments.

Although only 71.8% of large services adopt fully automated deployments, it already demonstrates a high degree of trust in deployment automation, considering that they are complex and hyperscale services serving billions of users.

|         | Continuous | Daily | Weekly | Biweekly/Monthly | ManualPick |
|---------|------------|-------|--------|------------------|------------|
| Regular | 54.6%      | 24.8% | 16.8%  | 0.4%             | 3.5%       |
| Large   | 16.0%      | 16.8% | 37.4%  | 1.5%             | 28.2%      |
| ML      | 99.9%      | 0.0%  | 0.0%   | 0.0%             | 0.0%       |
| Other   | 48.5%      | 26.3% | 19.0%  | 0.3%             | 5.9%       |

Table 1: Classification of pipelines based on their schedules.

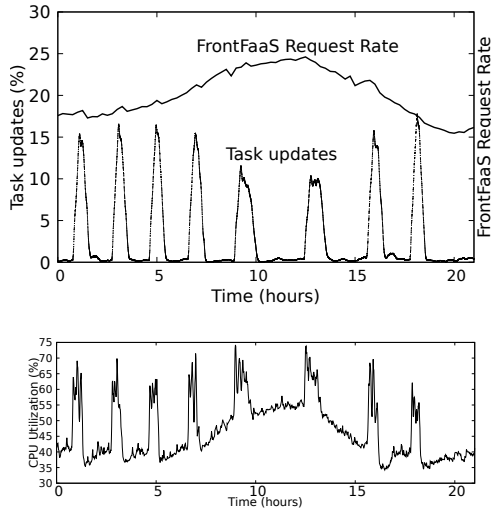


Figure 5: TaskControl slows down the deployment speed for FrontFaaS during peak hours.

One main reason for the remaining ones using manual validations is that they have highly complex health checks, making it difficult to achieve both a low false positive rate and a low false negative rate simultaneously, regardless of how the decision threshold for health checks is set.

Although the deployments of some services are already fully automated, they still prefer deployments on a fixed schedule (daily or weekly) over continuous deployment for several reasons. First, a failed deployment of a complex service may cause a partial outage in production, which can take hours to mitigate. Therefore, it is preferred to start the deployment at a fixed time in the morning to ensure that incident mitigation does not extend into the night. Second, although almost all services are fault-tolerant against task updates, they may experience degraded SLOs during a task update. For example, updating a ZooKeeper ensemble will trigger a leader re-election and result in delays in responding to client requests. These services prefer to avoid degraded SLOs caused by frequent deployments of every single code change.

### 6.3 Deployment Safety at Hyperscale

To ensure the safety of in-place updates, Twine’s TaskControl API allows services with special needs to exert precise control over their task updates. For example, FrontFaaS’ custom TaskController dynamically adjusts the deployment speed in order to safely and continuously deploy every three hours across more than half a million machines. Figure 5 illustrates what happens in a region that runs over 10,000 FrontFaaS tasks. The top figure shows the normalized request rate for FrontFaaS, along with the percentage of updates to FrontFaaS tasks in that region. The bottom figure shows the average CPU utilization of those FrontFaaS tasks. During the site’s peak hours, which occur between hours 8 and 15, FrontFaaS’ TaskController instructs Twine to reduce the number of task updates in order to prevent the temporary loss of too many

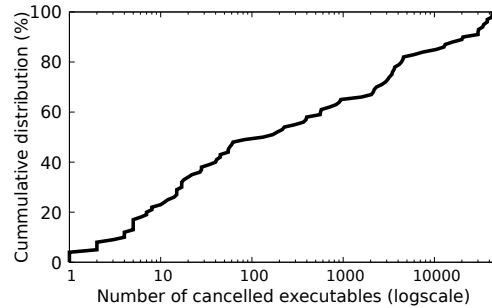


Figure 6: CDF of the number of executables canceled by the Bad Packet Detector (BPD) per reported bug.

|            | Regular services | Large services | ML models | Other pipelines |
|------------|------------------|----------------|-----------|-----------------|
| Successful | 27.5%            | 15.4%          | 33.7%     | 38.0%           |
| Failed     | 13.8%            | 20.9%          | 5.2%      | 4.2%            |
| Canceled   | 58.7%            | 63.7%          | 61.1%     | 57.8%           |

Table 2: Breakdown of the end states of releases.

tasks and avoid overloading the system. Consequently, it takes longer to complete a release during these peak hours. Despite the fluctuating load on the site and the load spikes caused by task updates (see the bottom figure), the overall CPU utilization remains below 75%. Without TaskControl, this level of application-specific adaptation and precise control is hard to achieve with other deployment tools.

To ensure the safe deployment of services that share a monorepo [9], which often entails complex code dependencies, developers can report bugs to Conveyor. The Bad Package Detector (BPD) automatically identifies the affected executables scheduled for deployment and cancels their releases. While only approximately one such bug is reported to Conveyor per day, their impact tends to be widespread. Figure 6 illustrates the number of executables affected by these bugs. Around 15% of these bugs impact over 10,000 executables. Certain bugs, such as those found in Meta’s RPC library [37], have the potential to impact every service. Due to the broad impact of these bugs, the BPD cancels approximately 14% of all executables scheduled for deployment. This extensive impact highlights the need for automated code dependency tracking in a monorepo to ensure deployment safety.

### 6.4 Deployment Failures

To understand how often deployments fail and why they fail, we analyze the failure data of releases and `deploy` actions.

#### 6.4.1 Release Failures

Over the three weeks of our evaluation, Conveyor generated millions of releases. Table 2 summarizes the end states of these releases. Release cancellations commonly occur when a new release supersedes an old release that was not yet deployed. While most releases are canceled, they are still highly valuable as they facilitate the execution of builds and tests, aiding in the early detection of bugs.

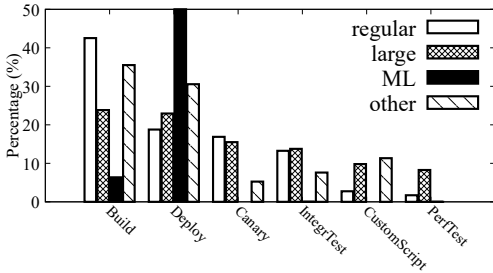


Figure 7: Breakdown of release failures by action types.

Although the release failure rate seems high, it actually indicates the effectiveness of deployment automation. Let’s consider a simple pipeline consisting of `build`→`test`→`deploy`. If a release fails during the `build` or `test` action, it means that the problem is detected early, preventing a faulty release from reaching production. This is precisely why, despite the high failure rate, developers maintain a high level of trust in deployment automation (§6.2).

We show the breakdown of release failures by action types in Figure 7. As an example, the value of “40%” for the “*regular*” bar in the `build` category does not mean that 40% of builds fail for regular services. Instead, it means that out of all failed releases for regular services, 40% of them failed while executing the `build` action. The value of the “*ML*” bar in the `deploy` category is 93.6%, but we cap it at 50% to make other bars more visible.

Figure 7 shows that, except ML models, the majority of release failures are detected by standard builds and tests (`canary`, `IntegrTest`, and `PerfTest`). Note that builds involve running unit tests and will fail if any tests do not pass. For ML models, the setup of parent-child pipelines between inference executables and models (§3.3) helps reduce failed releases. During the three weeks, nine failures occurred in the parent pipelines. Among them, eight were detected by unit tests, and one was detected by `PerfTest`. These failed parent pipelines did not trigger the execution of the corresponding child pipelines. Otherwise, the number of failed child pipeline releases would have increased by about 50%.

#### 6.4.2 Failures in Deploy Actions

Failures in `deploy` actions occur at the important stage of updating tasks and could lead to user-visible errors. The failure rates of `deploy` actions vary across pipeline types: regular services (5.4%), large services (6.2%), ML models (14.3%), and other pipelines (1.1%). In all cases, health check failure is the top reason for `deploy` failures, followed by update timeout, which is typically caused by too many unhealthy tasks during a deployment.

Figure 8 presents the breakdown of health check failures. While system-level metrics, such as CPU, memory, crashes, and RPC errors, can help identify many problems, “*AppSpecific*” metrics still play a significant role. Frequently used such metrics include the increase in the number of specific errors

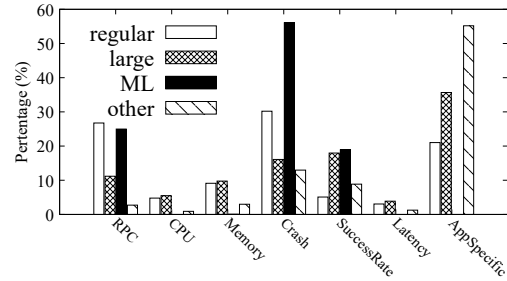


Figure 8: Breakdown of health check failures.

|                | Regular services | Large services | ML models | Other pipelines |
|----------------|------------------|----------------|-----------|-----------------|
| False negative | 0.92%            | 3.14%          | 0.012%    | 0.12%           |
| False positive | 11.5%            | 41.3%          | 0.0025%   | 17.4%           |

Table 3: False positives and false negatives of health checks.

returned to users, increase in the number of retries, decrease in correctness metrics, or changes in user engagement.

Developers often need to make a tradeoff between false positives and false negatives when using health checks to detect bugs because health anomalies can also be caused by other factors such as hardware failures or changes in workload. To approximate the rate of false negatives (i.e., bugs not being detected), we calculate the percentage of releases that finished successfully but were later reverted or patched by developers. However, since developers sometimes patch or revert a release for purposes other than fixing bugs (e.g., to do a quick test), these numbers should be viewed as an upper bound for false negatives. To approximate the rate of false positives (i.e., health anomalies in the absence of a bug), we calculate the percentage of `deploy` actions that reported health anomalies but were allowed to proceed by a human.

As shown in Table 3, except for ML models, the occurrence of false positives is significantly higher than that of false negatives. Notably, the rate of false positives for large services reaches as high as 41.3%. This is because health checks for large services are typically more intricate, and developers tend to use stringent health check thresholds to ensure release safety. When faced with health anomalies, they prefer to rely on manual investigations to determine whether to proceed with a release or not.

Figure 9 further presents the point at which a `deploy` action fails. The *progress* metric is calculated as the number of tasks that are supposed to be updated till the end of the failed phase divided by the total number of tasks to be updated in all phases. We exclude single-phase `deploy` actions from this figure, since their progress is either 0 or 1. Figure 9 reveals a bimodal pattern, where failures occur either very early or very late. While many bugs can be detected when only a small subset of tasks is updated, some bugs, such as subtle performance regression, can only be detected after they are deployed at scale. These kinds of bugs pose the most challenging problem for software deployment.



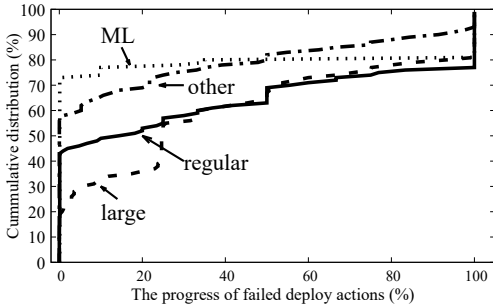


Figure 9: The progress of failed deploy actions.

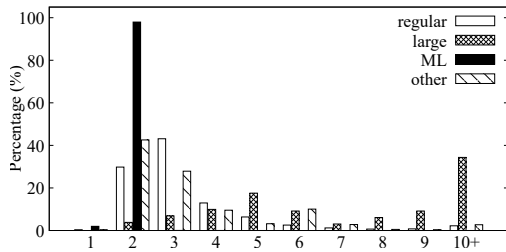


Figure 10: Number of actions per pipeline.

## 6.5 Pipeline Patterns & Recommendations

In this section, we analyze various aspects of pipeline statistics to gain a better understanding of how pipelines are used in production. Based on our findings, we provide best practices for pipeline design in §6.5.4.

### 6.5.1 Pipeline Configuration

Figure 10 shows that the average number of actions per pipeline varies: regular services (3.5), large services (12.2), ML models (2.0), and other pipelines (4.0). As expected, large services have much deeper pipelines. For ML models, all child pipelines (§3.3) use a uniform setup with two actions: build and deploy. The parent pipelines for inference executables have more actions such as PerfTest, but since the number of child pipelines is about 1,000 times larger than that of parent pipelines, the statistics here mainly represent those of child pipelines.

We further examine the popularity of different types of actions, represented as the percentage of pipelines that include at least one corresponding action type. As shown in Figure 11, build and deploy are the two most popular actions, as expected. We observe two distinguishing characteristics of large services. First, they are more likely to include tests (canary, IntegrTest, and PerfTest). Second, they rely more on human decisions (i.e., ManualPick) to determine whether to proceed.

### 6.5.2 Deploy Action Configuration & Runtime Statistics

To balance safety and speed, a deploy action often consists of multiple phases, each updating a subset of tasks. We observe a few popular patterns in the setup of deploy actions: 1) the “*super linear*” pattern, which updates a small percentage of

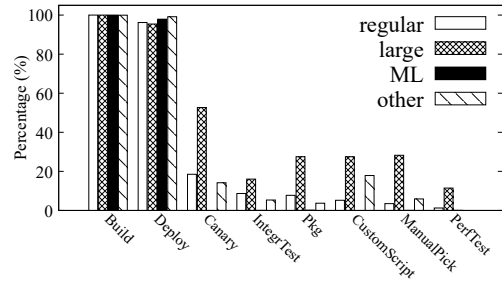


Figure 11: Percentage of pipelines that use a specific action.

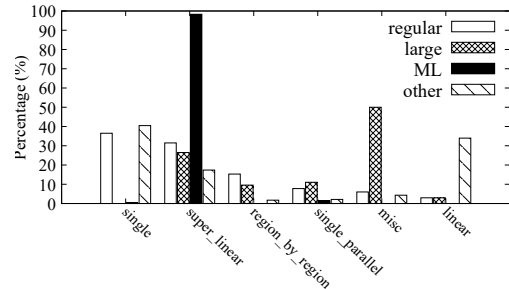


Figure 12: Breakdown of different deploy patterns.

tasks in the first phase and updates a higher percentage in later phases; 2) the “*single*” pattern, which updates all tasks in one phase; 3) the “*single parallel*” pattern, which uses one phase but updates multiple jobs in parallel; 4) the “*region-by-region*” pattern, which updates one region’s jobs per phase; and 5) the “*linear*” pattern, which updates the same percentage of tasks in each phase. By definition, these patterns are not exclusive. For example, a *region-by-region* pattern could be *linear* as well. To separate them, we use the following rule: *single parallel* > *single* > *region-by-region* > *linear* > *super linear*. It means that, if a *deploy* action meets more than one definition, we categorize it as the first one in the chain.

As shown in Figure 12, the setups are very diverse. Among regular services, simple ones prefer the *single* pattern and complex ones prefer the *super linear* pattern to balance safety and speed. Large services employ various *misc* patterns, with a concrete example shown in §6.5.3. ML models mostly use the *super linear* pattern. Among other pipelines, simple ones prefer the *single* pattern and complex ones prefer the *region-by-region* pattern for safety.

The execution time of deploy actions has a long-tail effect: for regular services, its P50 is 2.3K seconds and P99 is 86K seconds; for large services, its P50 is 2.0K seconds and P99 is 186K seconds (52 hours); for ML models, its P50 is 2.9K seconds and P99 is 9.8K seconds; for other pipelines, its P50 is 0.33K seconds and P99 is 15K seconds. We will elaborate on the long deploy time in §6.5.3.

Table 4 further decomposes deploy action’s execution time into four components: *task update*, *bake*, *preprocessing*, and *postprocessing*. Recall that the bake time is the time after all updates in a phase have been completed, during which Con-

|                | Regular services | Large services | ML models | Other pipelines |
|----------------|------------------|----------------|-----------|-----------------|
| Task update    | 21.2%            | 45.1%          | 39.2%     | 31.4%           |
| Bake           | 31.7%            | 25.1%          | 58.1%     | 29.6%           |
| Preprocessing  | 18.7%            | 2.9%           | 1.9%      | 5.4%            |
| Postprocessing | 28.4%            | 27.0%          | 0.9%      | 33.6%           |

Table 4: Time spent in different stages of `deploy` actions.

veyor periodically checks the health of the updated portion of a service. *Preprocessing* and *postprocessing* refer to custom operations performed before and after updating a subset of tasks, respectively. Examples include redirecting traffic and conducting end-to-end tests. Overall, Table 4 shows that operations other than task updates consume the majority of the time, emphasizing the importance of holistic optimization of the pipeline setup when deployment speed is a concern.

### 6.5.3 Real-world Example of Long Deploy Time

Large services that prioritize deployment safety may require multiple days to complete their `deploy` actions. To illustrate this, we present a real-world example of a foundational storage service at Meta that operates in every datacenter region. Its `deploy` action is configured to make progress on workdays from 9AM to 6PM without manual intervention. If the execution of the `deploy` action does not finish by 6PM, it will pause and continue on the next workday at 9AM.

Its deployment follows the *super linear* pattern for a few regions and then switches to the *region-by-region* pattern for the remaining regions. With over 20 phases in total, each of the early phases has a bake time of one hour, while the remaining phases have a bake time of 10 minutes. On average, task updates per phase take about 25 minutes. Taking into account the bake time, the early phases run for about 85 minutes each, while the remaining phases run for about 35 minutes each. Considering the total number of phases and their duration, the `deploy` action typically completes the early phases within one day and then resumes the next workday at 9AM. If everything goes smoothly, the deployment finishes on the second day. However, transient issues may cause it to retry and delay the completion time until the third day.

### 6.5.4 Recommendations for Pipeline Design

We recommend the following best practices for pipeline design. In general, we recommend using the *super linear* pattern as a starting point, as it provides a good balance between speed and safety. Moreover, we recommend including a phase in the middle of the pipeline to update all tasks within a region as opposed to never updating any whole region until the last phase. This is important because many services have regional dependencies, and certain issues, such as performance regressions, may only become noticeable when the code runs at the scale of a full region. Finally, we recommend scheduling deployments for the mornings of Monday to Thursday, so that developers have a full work day to troubleshoot any deployment issues. They may adopt continuous deployment after health checks and tests have matured.

## 7 Related work

There is a rich set of deployment tools, such as Spinnaker [42], AWS CodeDeploy [3], AWS CodePipeline [4], Azure Deployment Manager [46], Azure Pipeline [6], Google Cloud Build [17], Google Cloud Deploy [18], and CircleCI [13]. Cluster manager is also a well-studied topic. Examples include Kubernetes [22] and YARN [49] from open source, Borg [47, 50] from Google, and Protean [19] from Azure. Section 3 discussed advanced features that distinguish Conveyor and Twine from existing systems.

A number of prior works have studied and surveyed possible problems in software deployment [20, 23, 38, 41, 51], mostly based on open-source projects or individual case studies. As described in §3, the scale and diversity of the services at Meta have introduced many new challenges, such as in-place updates, handling complex code dependencies, fast deployment of large services like FrontFaaS, and deployment of complex ML models. Dedicated ML platforms such as AWS SageMaker can deploy models using the mirroring approach [35, 36], but they do not support in-place updates or the advanced model-deployment features described in §3.3.

Multiple works have focused on individual problems during deployment. For example, Gandalf tries to locate the problematic deployment after failures are detected [26]. ZebraConf tries to detect configuration updates that may cause compatibility issues [27]. Boyer et. al. [8] propose a declarative approach to update services, instead of the imperative approach used by most deployment tools, including Conveyor.

## 8 Conclusion

We presented the deployment scenarios, operational experience, and production data related to software deployment at Meta, along with the design and implementation of Conveyor. We demonstrated the feasibility of frequent and fully automated deployments supported by a single deployment tool for all services. Additionally, we presented novel techniques for in-place updates, analysis of code dependencies to prevent faulty releases, and the safe deployment of ML models.

## Acknowledgments

This paper presents nine years of work by past and current members of several teams at Meta, including Conveyor, Twine, Health Check Service, Configuration Management, Release Engineering, and Inference Platform. In particular, we would like to call out the current members of the Conveyor team who are not on the author list: Alex Rock, Arvind Gautam, Brian Fitzpatrick, Eddy Li, Haydn Kennedy, Jared Bosco, Jimmy Zeng, Marcos Pertierra Arrojo, Marija Trifkovic, Matthew Boardman, Rudy Pikulik, Mike Belov, Matthew Almond, Nippun Goel, Shawn Cui, and Martin Reichhoff. We thank Hui Lei, all reviewers, and especially our shepherds, Ding Yuan and Malte Schwarzkopf, for their insightful comments.

## References

- [1] Release early, release often. [https://en.wikipedia.org/wiki/Release\\_early,\\_release\\_often](https://en.wikipedia.org/wiki/Release_early,_release_often).
- [2] Blue/Green Deployments. <https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html>.
- [3] AWS CodeDeploy. <https://aws.amazon.com/codedeploy/>.
- [4] AWS CodePipeline. <https://aws.amazon.com/codepipeline/>.
- [5] AWS Lambda. <https://aws.amazon.com/lambda/>.
- [6] Azure Pipeline. <https://azure.microsoft.com/en-us/products/devops/pipelines/>.
- [7] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
- [8] Fabienne Boyer, Nol de Palma, Xinxiu Tao, and Xavier Etchevers. A Declarative Approach for Updating Distributed Microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, page 392–393, 2018.
- [9] Nicolas Brousse. The Issue of Monorepo and Polyrepo In Large Enterprises. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, pages 1–4, 2019.
- [10] Buck2. <https://buck2.build/>.
- [11] Emily Burns, Asher Feldman, Rob Fletcher, Tomas Lin, Justin Reynolds, Chris Sanden, Lars Wander, and Rob Zienert. Continuous Delivery with Spinnaker. <https://spinnaker.io/docs/concepts/ebook/>.
- [12] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.
- [13] CircleCI. <https://circleci.com/>.
- [14] eBPF. <https://ebpf.io/>.
- [15] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, Carlsbad, CA, July 2022. USENIX Association.
- [16] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.*, 30(4), November 2012.
- [17] Google Cloud Build. <https://cloud.google.com/build>.
- [18] Google Cloud Deploy. <https://cloud.google.com/deploy>.
- [19] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 845–861. USENIX Association, 2020.
- [20] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
- [21] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX, 2023.
- [22] Kubernetes. <https://kubernetes.io/>.
- [23] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82:55–79, 2017.
- [24] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 553–569, 2021.
- [25] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.
- [26] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-to-End Analytics Service for Safe Deployment in Cloud-Scale Infrastructure. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, NSDI'20*, page 389–402, 2020.

- [27] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 410–425, 2021.
- [28] Paul Marinescu. Autonomous testing of services at scale. <https://engineering.fb.com/2021/10/20/developer-tools/autonomous-testing/>, 2021.
- [29] Caroline Moss. Facebook Went Down And People Started Calling The Cops, 2014. <https://www.businessinsider.com/call-cops-when-facebook-is-down-2014-8>.
- [30] Guilherme Ottoni. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.
- [31] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.*, 8(12):1816–1827, August 2015.
- [32] Eric Raymond. The Cathedral and the Bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.
- [33] Chuck Rossi. Rapid release at massive scale. <https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/>, 2017.
- [34] Krzysztof Rzacca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.
- [35] Amazon SageMaker. <https://aws.amazon.com/pm/sagemaker>.
- [36] Amazon SageMaker UpdateEndpoint. [https://docs.aws.amazon.com/sagemaker/latest/APIReference/API\\_UpdateEndpoint.html](https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_UpdateEndpoint.html).
- [37] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.
- [38] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.
- [39] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120. IEEE, 2017.
- [40] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017.
- [41] Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. An Empirical Study of Architecting for Continuous Delivery and Deployment. *Empirical Software Engineering*, 24(3):1061–1108, 2019.
- [42] Spinnaker. <https://spinnaker.io/>.
- [43] Spinnaker rollout strategy for Kubernetes. <https://spinnaker.io/docs/guides/user/kubernetes-v2/rollout-strategies/>.
- [44] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatchalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.
- [45] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutorenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.
- [46] David Tepper. Introducing Azure Deployment Manager. <https://learn.microsoft.com/en-us/archive/msdn-magazine/2019/august/azure-devops-introducing-azure-deployment-manager>.
- [47] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In



*Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

- [48] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 805–816, 2015.
- [49] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.
- [50] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.
- [51] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306, 2018.

# Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases

Tamer Eldeeb  
*Columbia University*

Xincheng Xie  
*Columbia University*

Philip A. Bernstein  
*Microsoft Research*

Asaf Cidon  
*Columbia University*

Junfeng Yang  
*Columbia University*

## Abstract

Distributed on-disk database systems could either use an expensive commit protocol like two-phase commit (2PC) to guarantee atomicity, and suffer from *slow* distributed transactions, or forgo 2PC, which lead to weaker semantics, limitations to the programming model, or constrained scalability, making the system less *general*. We argue this compromise is no longer necessary within modern datacenters. Low latency 2PC ( $\sim 150$   $\mu$ s on Azure for 2PC over Paxos) can be achieved using low-latency storage for the relatively small transaction logs, fast RPCs, and careful protocol design. With fast 2PC, the data contention bottleneck for many transactions shifts from 2PC to reading the data itself from the relatively slow storage while holding transaction locks.

We present Chardonnay, a scalable, on-disk, multi-versioned transactional key-value store optimized for single datacenter deployments with fast 2PC. Chardonnay has a *general* interface supporting point reads, scans, and writes within multi-step strictly serializable ACID transactions. The key mechanism underlying Chardonnay's design is strongly consistent snapshot reads on commodity hardware, using a novel lock-free read protocol. Chardonnay uses this protocol to cheaply determine the read-write sets of queries, enabling Chardonnay to transparently prefetch data needed for a transaction prior to the execution of the transaction and the acquisition of locks. This enables Chardonnay to achieve *fast* transactions by minimizing contention, and avoids aborts due to deadlocks by ordering lock requests.

## 1 Introduction

The holy grail of distributed databases is to provide an abstraction of a single-server database that can run SQL ACID transactions at high performance while maintaining high availability. Recent work [27, 33, 47, 52, 57, 69, 81, 82, 84] shows that ACID distributed transactions

with strong isolation and consistency semantics can be made efficient and scalable within in-memory database systems. However, keeping all data in memory can be prohibitively expensive, especially for large applications, as DRAM's cost per GB is over 10–50 $\times$  more expensive than regular (e.g., TLC or QLC) NAND SSD [34].

Therefore, due to their significantly lower cost, many applications use distributed databases [11, 12, 29, 74], which store their data on disk-based storage engines such as RocksDB [8, 32, 58] or LevelDB [7]. The classic architecture for such systems [65], popularized by System R\* [59], is to shard the data horizontally across a collection of shared-nothing machines, and use a distributed commit protocol such as two-phase commit (2PC) [48] to ensure atomicity of distributed ACID transactions. Unfortunately, distributed transactions in these systems have significant performance limitations [17, 30, 42, 52, 57, 76, 81].

Due to these challenges, many scale-out on-disk systems avoid providing any multi-key ACID transaction support at all [26, 31], or limit it to local transactions accessing keys within a single machine or partition [23, 61]. Other systems offer support for distributed transactions, but forgo 2PC and sacrifice *generality* in one or more ways, e.g., by offering weaker semantics [16, 54, 78, 79], restricting the programming model [76], or employing an architecture that limits system scalability [13, 49, 87]. Nevertheless, due to strong developer demand [14], many popular SQL DBMSes now support general distributed ACID transactions [11, 29, 74], despite being a lot slower than local transactions. Table 1 shows the trade-offs made by various popular on-disk systems.

We argue that this compromise between performance and generality is no longer necessary within the modern datacenter. The high performance penalty of 2PC historically has been due to the high latency of RPCs and flushing log entries to disk. Fortunately, neither is the case any more. Modern datacenter networks are fast [22], and systems such as eRPC [46] have demonstrated that

| System                     | Serializable | Linearizable | General API | Distributed TX | High Contention         |
|----------------------------|--------------|--------------|-------------|----------------|-------------------------|
| Spanner [29]               | ✓            | ✓            | ✓           | Slow           | X                       |
| Calvin [76]                | ✓            | ✓            | X           | Fast           | ✓                       |
| FoundationDB [87]          | ✓            | ✓            | ✓           | Fast           | X                       |
| Hyder [21]                 | ✓            | X            | ✓           | N/A            | X                       |
| Aurora (Multi-Master) [78] | X            | X            | ✓           | N/A            | Partitionable Workloads |
| Chardonnay                 | ✓            | ✓            | ✓           | Fast           | ✓                       |

Table 1: Comparison of representative on-disk distributed database systems.

RPCs can run at single-digit  $\mu$ s latency within the data-center even without using RDMA. Additionally, storage devices based on low-latency SLC NAND [10] or 3DX-point [1] also provide single-digit  $\mu$ s latencies [6, 9, 10], making them ideal for persisting database logs.<sup>1</sup> Furthermore, many recent frameworks [45, 66, 83, 85, 86] fully or partially bypass the Linux I/O software stack, further boosting I/O performance.

This leads us to revisit the assumption that 2PC is the primary bottleneck inherent in scale-out on-disk database system designs. However, using a fast 2PC protocol reveals new bottlenecks. As we show in §4, even *eliminating the entire latency of the commit protocol is not sufficient* to achieve good performance for high-contention workloads, because transactions frequently hold locks while fetching cold items from storage. Therefore, the data contention bottleneck shifts to reading the data from disk, since reading data from a typical SSD can be orders of magnitude slower than the network.

We present Chardonnay, a distributed multi-version transactional key-value store that is deliberately tailored for this new era of fast 2PC. Chardonnay is designed for single-datacenter deployments, since cross-datacenter 2PC latency would be high. It supports point and range reads, as well as writes, within classical multi-step strictly serializable ACID transactions, making it suitable as the storage engine for a SQL database (e.g., similar to CockroachDB [74]). Chardonnay uses the classic shared-nothing architecture<sup>2</sup> and uses strict two-phase locking (2PL) [37] to guarantee strict serializability [43] for read-write transactions, as well as 2PC to ensure atomicity for distributed transactions.

The core insight of Chardonnay is that fast RPCs enable strictly serializable lock-free snapshot queries within the datacenter in a *general* fashion, i.e., without using specialized clocks, limiting scalability, or weakening the performance and semantics of read-write transactions. Low-latency, high-throughput RPCs are key to allow all committing transactions in Chardonnay to

cheaply read a counter, called the *epoch*, that serves as a global serialization point. The system increments the epoch periodically, independent of transactions, so unlike designs with a centralized sequencer [18, 87], maintaining the epoch can be distributed and highly scalable. The main challenge is that unlike systems with a single global log or coordinator, Chardonnay uses one log per partition, so it cannot enforce global epoch ordering of commits. Instead, we co-design the snapshot read and commit protocols to guarantee their equivalence to epoch ordering (§6). The idea is rather simple: Snapshot queries may block waiting for write locks to be released (once) for correctness, but they do not acquire any locks, so they do not contend with the read-write transactions.

Beyond the direct benefit of efficient, lock-free read-only queries, this enables two important benefits, as Chardonnay leverages this snapshot read protocol to optimize the execution of read-write transactions. First, Chardonnay runs the user’s transaction in a *dry run* mode using the snapshot protocol to (approximately) compute and prefetch the transaction’s read set, which in the vast majority of cases allows Chardonnay to shift the work of reading cold data from storage outside of the contention period of the transaction. Second, since read and write sets can be efficiently computed using the snapshot protocol, Chardonnay also uses them to plan the locking scheduling in a manner that avoids deadlock aborts.

At the systems and design level, our main contribution is Chardonnay, the first (to our knowledge) on-disk system that achieves high performance for both low and high-contention workloads, without sacrificing strong semantics, restricting the programming model, or limiting scalability. The novel mechanisms introduced in Chardonnay are:

- Novel lock-free snapshot read protocol:** Chardonnay uses fast RPCs to guarantee strict serializability without relying on specialized hardware, synchronized clocks, making assumptions about clock skew, or limiting scalability.
- Automatic prefetching:** Chardonnay leverages the snapshot protocol to do a “dry run” of the query, which loads and pins all the keys accessed by the

<sup>1</sup>It is of course possible to store the entire database on such devices, but they cost significantly more than commodity SSDs.

<sup>2</sup>Which, we posit, has aged like fine wine.

transaction to main memory. This allows Chardonmay to avoid waiting for data read from slow storage while holding locks. Unlike similar schemes introduced by prior work [3, 75, 76], Chardonmay's prefetching mechanism works for scans, and neither requires changes to the user code, nor incurs significant additional latency or contention.

3. **Lightweight deadlock avoidance:** By computing read and write sets in advance, Chardonmay avoids deadlocks by determining the lock acquisition order.

Collectively, these techniques enable Chardonmay to have excellent performance under high contention. Indeed, as we show in §9, Chardonmay's throughput under extremely high contention is only 15% lower than under extremely low contention. In contrast, the throughput of a baseline System R\*-style system (even utilizing fast 2PC) drops by over 85%. The dry run phase adds overhead which is largely wasteful for low contention workloads, but we consider this a worthwhile trade-off, and we allow disabling dry runs on a per transaction basis.

A general takeaway is that within on-disk systems, the availability of fast datacenter RPCs makes distributed and multi-core system designs look increasingly similar. Some of our ideas (epoch-based versioning) are inspired by multi-core database systems [77]. This unlocks the potential for adopting additional insights from multi-core single-node systems in a distributed setting. The flow of ideas can also go in the other direction: while distributed transactions were our primary motivation when designing Chardonmay, the challenge of high contention is not unique to distributed transactions, and in fact many single-node database systems run with low isolation precisely to mitigate this issue [15]. Our results show that Chardonmay's techniques can be useful for them too.

## 2 Background

This section discusses transaction semantics and commit protocol performance in distributed database systems.

### 2.1 Strict Serializability

Strict Serializability [43] (also known as External Consistency [29]) is considered the gold standard of distributed transaction semantics. It is the combination of the following two properties [69]:

- **Serializability:** every execution is equivalent to some serial ordering of committed transactions.
- **Linearizability:** if transaction A commits before transaction B starts, then A should precede B in the equivalent serial ordering.

### 2.2 2PC Recap

Two-phase commit (2PC) is a classic commit protocol with many variants [48]. The basic flow works as follows: after a transaction finishes execution on multiple *participant* servers or shards, a *coordinator* starts the first phase by issuing *Prepare* RPCs to all participant. Each participant can vote yes or no in response to the RPC, where a yes vote is a promise by the participant that it will not unilaterally abort the transaction and will be able to (eventually) commit the transaction when asked. Before voting yes to a Prepare RPC, the participant typically persists all of the transactions writes to a durable log so it can recover from any failures. If any participant votes no (or never responds due to failures or timeouts), the coordinator aborts the transaction. Otherwise, it logs the decision to commit to durable storage and then runs the second phase of the protocol by issuing *Commit* RPCs to the participants so they can apply the transaction and release locks. A well known problem of 2PC is that it is *blocking* [20, 70], wherein the failure of the coordinator at inopportune moments prevents the participants from making progress. This can be addressed by replicating the coordinator state for availability [17, 29, 40].

### 2.3 The Penalty of 2PC

2PC traditionally incurs a significant performance overhead for two main reasons. First, it requires at least two network round trips and two synchronous log writes to persistent storage per transaction [41, 57], which incurs network and storage I/O overhead, as well as CPU usage by the TCP/IP stack [81]. For example, typical 2PC commit latency within a single datacenter in systems like Spanner is in the double digit milliseconds [29], which puts a hard upper bound of less than 100 TPS on transactions that update a write-hot record. Second, the coordination necessary to guarantee isolation can significantly decrease concurrency, leading to performance degradation, as well as high abort rates [15]. This increased contention due to 2PC is particularly harmful for short transactions common in OLTP workloads, due to the high latency of the commit protocol relative to the time it takes to execute the transaction logic [76]. The impact of contention is evident in locking-based concurrency control schemes such as 2PL, but optimistic concurrency control (OCC) schemes are also not immune, and can in fact perform worse under high contention [41, 51, 82].

## 3 Requirements

We now define Chardonmay's stated objective, *fast and general* transactions for on-disk databases, in more detail. Fast encompasses the following requirements: First,



latency for short OLTP transactions should be low (hundreds of  $\mu\text{s}$ ) regardless of whether it is single partition or cross-partition; hence the performance penalty of distributed transactions should be relatively small. Second, the system needs to support long-running read-only queries efficiently, without impacting OLTP read-write transactions. Finally, the system should be able to maintain high throughput for both low and high contention workloads. General means providing a general, unrestricted programming model and API (e.g. capable of supporting a full SQL layer) and the highest level of semantics (i.e. strict serializability) without imposing overall scalability limits or using specialized hardware.

#### 4 Measuring Contention Footprint

Data contention is a major issue for traditional on-disk shared-nothing distributed database designs. Most real-world workloads have low contention most of the time, but occasionally a small number of extremely hot data items appear, significantly degrading overall throughput [39, 76]. Other workloads are characterised by high skew such that a small portion of the database receives a majority of the load. For example, half of the NYSE trades happen on 1% of the symbols, and breaking news can cause a sharp spike in trades on a small group of symbols [68]. Indeed, data contention is a bottleneck that hinders truly scalable transaction processing, even in RDMA-enabled in-memory distributed database systems [82], and on multi-core single-node systems [62].

Following the terminology of Calvin [76], we define a transaction’s *contention footprint* as the total duration from the instant the transaction acquires its first lock until it releases its last lock. In this section we use YCSB [28] to study the contention footprint of simple, single operation transactions in System R\*-style systems. To this end, we built two simple baseline systems based on the System R\* architecture on top of RocksDB, using its transaction and 2PC support in our experiments:

- **Baseline-Slow.** The client invokes database functions using (slow) gRPC [5]. Both the write-ahead log (WAL) and the database are placed on a directly attached SSD.
- **Baseline-Fast.** Uses (fast) eRPC (with FlatBuffers [4] for serialization format) instead of gRPC, and the WAL is put on an emulated fast NVMe device.

Our baseline implementations ignore crucial practical considerations (such as replicating coordinator state for high availability to deal with the well-known 2PC blocking problem), and transactions more complicated than a single read or write. Therefore, our results underestimate the contention footprint. Nevertheless, they are instructive. All our experiments run on Microsoft Azure VMs. The entire key universe is assigned to a single shard. We

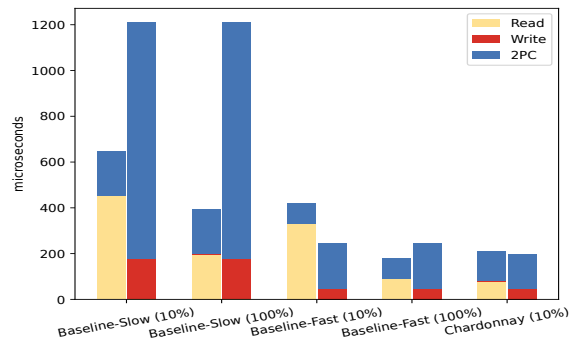


Figure 1: Contention footprint of YCSB read (left bar) and write (right bar) transactions. % represent the proportion of the data in DRAM. Chardonnay achieves a similar contention footprint to fully in-memory (“Baseline-Fast 100%”) with only 10% of its data in DRAM.

run YCSB-A with 50% point reads and 50% point writes with uniform random distribution. All experiments use one client with 5 threads, which runs on a dedicated VM in the same Azure region as the server. To control the amount of DRAM used by the system, we disable the OS page cache and vary the size of the block cache, which is RocksDB’s read cache. We run a full 2PC at the end of each transaction, including in the case of reads, to measure transaction overhead, even though technically 2PC is not needed since there is only one shard. Read transactions release locks during the Prepare phase, so the Commit phase does not contribute to their contention footprint. For durability, Calls to Prepare and Commit always wait for the write to be flushed to storage.

We show how the average latency of read and write operations each contribute to the contention footprint in Figure 1. On Baseline-Slow, the bulk of the contention footprint comes from running 2PC. On Baseline-Fast, the latency of 2PC is significantly lower due to the fast RPC library and fast log storage. The yellow bars show that the contention footprint of read transactions is much higher when only 10% of the dataset is in main-memory, since the majority of reads have to fetch data from SSD storage. Write transactions (red bars) are not much affected by the available DRAM, since writes are buffered in-memory (at the server) until the Prepare phase where they get written to the WAL.

We deduce two takeaways from this simple experiment. First, with a modern RPC library, fast intradatacenter network, and small amount of fast NVMe storage, distributed databases can significantly reduce 2PC latency. Second, once the latency of 2PC is reduced, the data contention bottleneck becomes reading the data needed by the transaction from the relatively slow SSD.

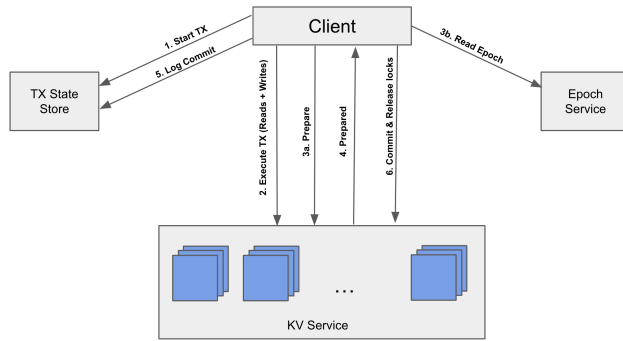


Figure 2: Transaction Lifetime in Chardonnay.

## 5 Architecture

Chardonnay has four main components:

1. **Epoch Service.** Responsible for maintaining and updating a single, monotonically increasing counter called the **epoch**. The epoch service exposes only one RPC to its clients, which returns the latest epoch. Reading the epoch serves as a global serialization point for all committing transactions. The epoch is used to assign transaction timestamps at commit time and is essential for our lock-free strongly consistent snapshot reads (§6). The epoch is only read, not incremented, by each transaction.
2. **KV Service.** The core service that stores the user key-value data. It uses a replicated shared-nothing range-sharded architecture similar to other modern System R\*-style systems [11, 29, 74].
3. **Transaction State Store.** Responsible for authoritatively storing the transaction coordinator state in a replicated, highly-available manner so that client failures do not cause transaction blocking. We chose to store the transaction state separately from the user’s key-value data to enable 2PC latency optimizations, which we describe in appendix A.1.
4. **Client Library.** Applications link this library to access Chardonnay. It is the 2PC coordinator, and provides APIs (Figure 3) for executing transactions.

Figure 2 illustrates how the components interact during the lifetime of a transaction. The basic flow of a read-write transaction is almost the same as in a classic shared-nothing System R\*-style system, except we add step 3b to read the epoch in parallel to the Prepare phase.

### 5.1 Epoch Service

The epoch service is a Multi-Paxos replicated state machine maintaining a single counter, the *epoch*. One replica is designated leader. It increments the epoch at a fixed configurable time interval (e.g., 10 ms) by appending an entry to the Paxos log so it is durably replicated.

It exposes one RPC, *read-epoch*, which returns the value of the epoch. The system maintains the invariant:

**Monotonic Epoch Invariant:** If a *read-epoch* call returns a value  $e$ , then all subsequent *read-epoch* calls must return a value greater than or equal to  $e$ .

We cannot rely on simply reading the value from the leader replica, since a leader might lose its status without realizing it for a while. It is possible to run the client RPCs through the Paxos state machine. However, since each committing transaction reads the epoch, this would be too costly. Instead, we consider the epoch updated when it is applied to the state of a majority of replicas, not just when it is appended to the log. The client sends read RPCs to all replicas and considers the current epoch value to be the one returned by a majority of the replicas. If no value has a majority, the client retries the read.

There is a trade-off in choosing the epoch advancing interval. It needs to be long enough compared to typical transaction duration that the value is usually read from the CPU caches of replicas, and without requiring retries due to no value having a majority. On the other hand, if it is too long, it adds to linearizable snapshot read-only transaction latency, as we explain in §6. We find that advancing the epoch once every 10 milliseconds works well in our experiments.

A single core can support tens of thousands of clients and serve up to millions of eRPC calls per second [46]. Furthermore, the client library batches multiple *read-epoch* calls from multiple concurrent transactions into a single RPC. Since each RPC does very little work (reads a word from main memory that is usually cached), we expect this design to be sufficient for all practical purposes. Nonetheless, in the interest of generality we show how to scale-out the epoch service in appendix A.3.

### 5.2 KV Service

The key universe is partitioned into disjoint contiguous subsets called **ranges**. Each range is assigned to a number of range servers (e.g., three) and is comprised of a database and a WAL that is implemented via Paxos. The WAL is placed on a fast NVMe device for low latency, while the database is stored on commodity SSD storage. One of the range replicas is designated as a *leader*, which holds a leader lease. It maintains a lock table to implement two-phase locking, using existing range locking techniques [50, 55]. All reads and writes go through the leader.

To simplify the description in this paper we will assume the ranges and replica-to-server assignments are static, although in practice ranges need to be moved, split and merged to balance load effectively. This can be accomplished using well-known techniques [23, 26, 29, 74], which we leave for future work.

### 5.2.1 Leader Selection and Disjointedness

Each range should have a designated leader replica that holds the leader lease. The leader selection is piggy-backed on the Paxos log implementation, i.e., a replica attempting to acquire the leader lease does so by appending a lease acquisition entry to the Paxos log. This log entry includes, among other information, the identity of the replica that is the lease holder, an *epoch interval* entitling the replica to leadership status as long as the epoch (maintained by the epoch service) falls within this interval, and *leader sequence number*, which is incremented whenever a new replica becomes the leader (but not when an existing leader renews its lease). The leader returns the sequence number to the client on every request, so the client can detect leadership changes and abort the transaction if the transaction observes two different leaders for the same range. When a leader is renewing its lease or a new leader is taking over, they read the epoch from the epoch service and set the upper interval ahead of the current value (by 100 in our prototype); it is important that the upper end is not too far ahead of the epoch, because this would effectively prevent other replicas from taking over if the leader goes down, until the true epoch catches up.

To prevent two replicas from acquiring leases with overlapping epoch intervals, a lease acquisition entry by a replica includes a copy of the lease believed to be the most recent. Other replicas will reject a replica's attempt to get the lease if they are aware of a more recent lease having been granted. This guarantees that at any point in time there is at most one leader for any range, and that only one range leader can successfully prepare transactions for an epoch. We call this the *Leader Disjointedness* invariant. In §5.4 we explain how we use it to validate transaction locks, and later in §6 we describe its role in the correctness of our lock-free snapshot reads.

## 5.3 Transaction State Store

The transaction state store is responsible for storing the state of active transactions in the system in a fault-tolerant, replicated manner, to mitigate 2PC blocking.

Each transaction can be in one of the following states: *Started*, *Committed*, *Aborted*, and *Done*. Note that being *Prepared* is not of concern here. We use the well-known presumed abort optimization [59], meaning that the service replies *Aborted* to a participant's inquiry about the state of a transaction unknown to the service. Being in *Done* state means that all transaction participant *ranges* have learned about the commit outcome of the transaction so that the service can safely forget about it.

The service is hash-partitioned by transaction id. Each partition is assigned to (typically) three servers. We do

```
class IChardonnay {
public:
    Transaction* start();
    std::string get(Transaction *tx, const std::string &key);
    std::vector<std::string> scan(Transaction *tx,
                                const std::string &lowerBoundInc,
                                const std::string &upperBoundExcl);
    std::string put(Transaction *tx,
                    const std::string &key,
                    const std::string &val);
    void del(Transaction *tx, const std::string &key);
    void abort(Transaction *tx);
    bool commit(Transaction *tx);
};
```

Figure 3: Simplified Chardonnay Client API

not need a per partition log to order transactions, since transactions are already ordered by 2PL. Instead, within a partition, each transaction state is represented as its own Multi-Paxos replicated log, which can have at most 3 entries. Position 0 always contains the Started entry, position 1 can either contain Committed or Aborted, and position 2 is to record Done state. This unusual design is key to a 2PC latency optimization that we describe in appendix A.1.

Recall that the client in Chardonnay acts as the 2PC coordinator. If the client crashes after starting the *Prepare* phase and before completing the transaction, the participant ranges need to determine whether to commit or abort. A KV Service range leader will attempt to put an Abort entry in the transaction state log (in position 1). If it succeeds, it can safely abort the transaction. The transaction state store is the source of truth regarding a transaction outcome. If the KV range leader successfully installs an abort decision for the transaction with the TX state store, a slow client cannot then succeed in committing it at a later point. Alternatively, after running the Paxos state machine, the KV range could learn that the client already put a Commit entry in that log position, in which case it can safely apply the transaction.

## 5.4 Client

The client provides an interface for users to access the database, and also acts as the 2PC coordinator in Chardonnay. After the transaction finishes execution, the client reads the epoch from the epoch service in parallel to issuing Prepare RPCs to participant range leaders. Each leader that accepts the Prepare request responds with a *Prepared* message that includes the epoch interval on its lease. The client then checks that the epoch it read falls within the lease's epoch interval of every participant, and if not, aborts the transaction. This is necessary to maintain the leader disjointedness invariant. If all the participants prepare successfully and the lease validations pass, the client then calls the transaction state store

to record the transaction's commit durably. The Commit record includes the participant ranges and the value of the epoch. Finally, the client calls the participant range leaders to notify them of the commit so they can record it locally and release all the locks. Transactions in Chardonnay must wait until the transaction Commit is recorded before releasing any locks, for the correctness of snapshot reads (§6). This implies that even read locks for successfully prepared transactions have to survive leader changes and thus must be logged in the WAL during the Prepare phase.

Many, if not most transactions only touch keys within a single range, so they do not need 2PC. First, the client reads the epoch. Then, it sends a Commit message to the leader, which checks that the epoch falls within the lease's epoch interval. If so, the leader appends to the WAL and if successful, returns success. If not, it aborts.

## 6 Snapshots

This section describes Chardonnay's multi-versioning and snapshot read protocols. Snapshot reads are essential to efficiently support read-only queries. They also underpin the techniques described in subsequent sections. Queries have to be declared as read-only from the start; a transaction that starts normally without this declaration but only performs reads is treated as a read-write transaction by the system, and does not utilize the lock-free snapshot read algorithm.

### 6.1 Versioning

Each user record has a key  $k$  and one or more versions stored in the database. The key for each version is the pair  $\langle k, \text{VID} \rangle$ , where VID (version ID) is determined as follows. Its prefix is the value of the epoch that the client reads in parallel to running the Prepare phase of 2PC. A counter (starting from 1) is appended to the epoch to distinguish writes by different transactions in the same epoch. A transaction chooses a single suffix that makes its VID greater than that of the existing VIDs in its write set. Deletes need to have versions as well, so they appear as tombstones. For convenience, the system also stores an unversioned record with just the key  $k$  which holds the latest value and is updated in place.

### 6.2 Read Algorithm

**Epoch Ordering Property:** There exists an equivalent ordering to the transaction ordering enforced by Chardonnay's strict 2PL such that for all pairs of committed transactions,  $T_1$  with an epoch  $e_1$ , and  $T_2$  with an epoch  $e_2$ , if  $e_1 < e_2$ , then  $T_1$  precedes  $T_2$ .

We present a proof sketch of this property in appendix A.2. The epoch ordering property ensures that epoch boundaries are consistent points in the serial order and appropriate for serializable snapshot reads, i.e., a transaction can get a consistent snapshot as of the beginning of the current epoch  $e_c$  by ensuring it observes the effects of all committed transactions that have a lower epoch. Suppose all the transactions with an epoch  $e < e_c$  have committed. Reading a user key  $k$  as of the start of epoch  $e_c$  translates to reading the value of key  $\langle k, \text{VID} \rangle$  such that VID is the largest value  $< \langle e_c, 0 \rangle$  in the database. Hence, the snapshot read algorithm would simply work by reading the epoch  $e_c$ , then reading the appropriate key versions.

The main challenge is ensuring that the snapshot is complete, i.e., no more transactions will be committing with an epoch below  $e_c$ . Any transaction that has not started to prepare is guaranteed to have an epoch of at least  $e_c$ , by the monotonic epoch invariant.

The problem is prepared (or preparing) transactions that are not yet known to have committed. Fortunately, any such transaction that could possibly commit writes must *already* be holding write locks at the current range leader. More formally, the transaction must be holding write locks on any replica whose leader lease's epoch interval upper end is above  $e_c$ . To see why this holds, suppose a transaction  $T$  with an epoch  $e_T < e_c$  has completed the Prepare phase but not the Commit phase. Recall from §5.4 that the client acting as  $T$ 's coordinator receives the epoch range of the lease from the range leader it used to perform the Prepare, and checks whether  $e_T$  falls within that epoch range. If it did not, then the client aborts the transaction so it cannot possibly commit. Otherwise, recall that transactions do not release any locks until the commit phase, including across leader changes. Therefore, it must be that the locks are held on the leader whose lease's epoch range contains  $e_c$  (and by the leader disjointness invariant, there can be at most one such replica), and any subsequent leader replica. A similar argument shows why the same holds for transactions that started but have not finished the Prepare phase. Hence, the read algorithm first reads the current epoch  $e_c$  (once per transaction), ensures it is below the upper end of the leader's epoch interval, and *waits* for the current holders of write locks (if any) on its read set to release these locks before executing the reads. The read is not attempting to *acquire* locks, so it does not contend with read-write transactions.

The algorithm as described so far does not guarantee linearizability, because a transaction  $T$  would not observe the effects of transactions in epoch  $e_c$  that committed before  $T$  started. If desired, ensuring linearizability is easy at the cost of some latency; after  $T$  starts, it waits for the epoch to advance once and then use the new epoch.



## 6.3 Garbage Collection

Chardonnay must periodically remove old record versions to avoid running out of space. Chardonnay uses the lower end of its range leader lease's epoch interval to determine which versions are no longer needed and can be garbage collected. There is a background job running on each range replica that removes versioned records (other than the newest version of a record) whose epoch is less than a delta from the lower end of the epoch interval. A snapshot read must validate that its epoch value lies within that delta from the lower end of the interval after executing its reads, to avoid reading an incomplete snapshot due to versions being deleted. In our experiments we configure the delta to be 6000, so that versions are kept at least for roughly one minute before they are GC'd. Additionally, since snapshot reads only happen at epoch boundaries, when a new version of a record is inserted, if it has the same epoch as the previous version then that previous version is immediately deleted. This optimization significantly reduces the number of versions maintained for records that are updated very frequently (i.e. highly-contended records).

## 7 Prefetching

Our experiments in §4 show that with fast 2PC, reading from slow storage becomes a primary cause of a transaction's contention footprint. Hot contended records will typically be cached in the database's memory. However, this does not fully address the issue because a transaction might access hot records along with other cold records that are not good candidates for caching. There are several well-known techniques to work around this problem [20]. For example, the programmer could manually prefetch records before executing the transaction. Another technique is to ensure hot records are the last to be accessed. This is beneficial because it minimizes the execution time during which access to the hot record causes a conflict. Unfortunately, these are not always applicable, and they push a lot of complexity to programmers.

We could require the programmers to label their queries with the read set. Then the system can prefetch the records (i.e., key-value pairs) identified by those keys to memory before executing the transaction and pin them until the transaction finishes, so that no time is spent reading from slow storage while locks are held. However, this scheme restricts the programming model, and is incapable of supporting *dependent queries*, that is, ones whose read set cannot be determined prior to executing the query [76]. This contradicts Chardonnay's goal of a general programming model (e.g., supporting SQL).

Instead, Chardonnay *transparently* uses the client's code to first execute the query in a lock-free, *dry run*

mode to load the read set to memory, then executes normally with 2PL.

It is of course possible for the read set to change by the time the actual transaction executes. One reason is that only the *values* of one or more records change due to a write by another transaction. Chardonnay handle this correctly and with no performance penalty, by reflecting the changes in its prefetching buffer (§7.3). The other possibility, in the case of dependent queries, is that the set of *keys* itself changes, so the transaction has to read some keys that had not been prefetched and pinned. This does not pose a correctness problem but may cause a transaction to read additional data from disk while it is holding the locks, and thereby increasing its contention footprint. Fortunately, prior work has shown this seldom happens in real-world workloads [76]; dependent queries are commonly ones that must read from a secondary index to identify their full read and write sets. Since secondary indices are fairly expensive to modify, they are seldom kept on fields whose values are updated very frequently. One example of such transactions is the "Payment" transaction of the TPC-C benchmark. Since the TPC-C benchmark workload never modifies the index on which Payment transactions' read and write sets may depend [76], the set of keys read by a Payment transaction never changes between the dry run and the execution.

One additional benefit of strongly consistent dry run queries is that if the application logic aborts the transaction on its own, there is no need to perform the actual execution. On the other hand, using dry run queries has two main disadvantages. First, it adds to the query latency, although this additional latency does not contribute to the contention footprint. Second, it requires executing the transaction logic twice before committing. While OLTP read-write transactions tend to be small, this could still be wasteful if the transaction is compute-intensive, particularly in low contention cases. The user can disable dry run queries by using a different API. In the future, we plan to explore automatically deciding when to do prefetching based on the characteristics of the workload.

### 7.1 API

The API shown in Figure 3 is more suited to user-interactive transactions (e.g., a user executing a multi-statement SQL transaction at a console, examining intermediate results before writing more queries). To use prefetching, a slightly different API is used to start the transaction where the caller passes a function that executes the transaction logic, i.e.,

```
<typename T>
T run(std::function< T( Transaction* ) > query)
```

Within the function, the code can freely call the read, scan, or write APIs using the transaction object that gets

passed. There are essentially no restrictions on the code inside the function, even though in practice it would have no side effects beyond the transaction's writes to the database itself. This does not add any unusual restrictions; any transaction might have to abort, and side effects outside of the transaction cannot be rolled back.

## 7.2 Semantics

The dry run query runs under snapshot isolation using our snapshot read mechanism that we described in §6, and can be configured to be strictly serializable if desired. Running under a lower isolation level such as *read committed* [19] could be problematic because it exposes the programmer's code to anomalies that would not happen in the serializable execution. This might cause the client's code to abort the transaction, prematurely ending prefetching, or worse, crash. Therefore, we do not use a lower isolation level because prefetching should be transparent to the user.

## 7.3 Design

Each range leader maintains a *prefetching buffer* to store a transaction's read set's records in main memory. The prefetching buffer tracks which records are in main memory and allows transactions to request pinning keys. Any committed write to a pinned record updates the value in the buffer, so that it becomes a write-through consistent cache for pinned records, and any transaction that needs to read a pinned key can just get its value from the cache and not have to go to the database.

To efficiently support range-queries, the prefetching buffer tracks key ranges not just individual keys; if a key range is pinned to the buffer, and a new transaction inserts (or deletes) a record whose key falls within that range, that new record is pinned too. Hence, a transaction that sees a range is pinned to the buffer can satisfy a range read from the cache without going to the database.

When a transaction is running in dry run mode, it reads the committed, snapshot value from the database without acquiring any locks, requests pinning the key (or key range), and then returns the committed value to the client to continue executing the query. In most cases both the snapshot and latest versions can be read using a single IO, so this does not typically increase the IO overhead. Writes made by the dry run query never make it to the KV Service, and are discarded at the client after the dry run. After the dry run completes, the client library reruns the transaction in normal mode. When that transaction finishes (i.e. commits or aborts), the keys are unpinned and become eligible for eviction.

It is possible that a request to pin a record cannot be satisfied because the range leader has run out of mem-

ory. In this case the dry run query could be delayed until memory frees up, or just be aborted. This serves as effective admission control prior to acquiring any locks. Some care is needed to avoid a potential live-lock situation, but in the worst case transactions can skip prefetching.

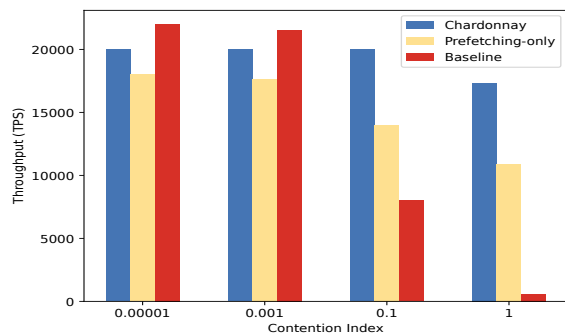
## 7.4 Handling Resource Contention

Dry run queries execute most of the transaction logic in Chardonnay, so that when the actual transaction executes it only needs to perform minimal work. However, if we are not careful, the activity from dry run queries and other background tasks such as garbage collection and compaction can compete with transactions for resources on the machines running the KV-service ranges. As a side effect, this could slow down the lock-acquiring transaction and increase data contention. Therefore, we dedicate resources on each machine to transactions to ensure they are insulated from lower-priority activity that does not hold locks.

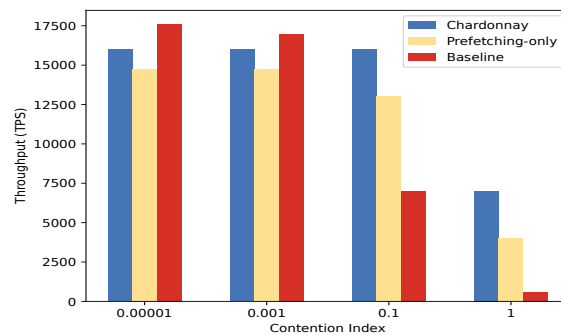
## 8 Deadlock Avoidance

Since Chardonnay uses 2PL, it has to deal with the problem of deadlocks. An easy solution is transaction timeouts, since they are needed anyway to deal with various possible failures. Unfortunately, a deadlocked transaction would be holding locks for the entire timeout duration before these locks are released. Another popular choice is a deadlock prevention scheme such as Wait-Die or Wound-Wait [64], but they can be too conservative (i.e., aborting transactions that are not deadlocked) which can become problematic under high contention. A more common choice in practice is deadlock detection [63] via detecting cycles in the wait-for graph [38, 72] and breaking the cycle by choosing a transaction to abort. This requires significant overhead for maintaining the global wait-for graph state, and potentially frequent aborts.

By making all transactions acquire their locks in the same order, we can prevent deadlocks. In Chardonnay, the read and write sets of the transaction are (approximately) computed by dry run queries, prior to acquiring any locks. We acquire the locks in ascending key order prior to actually executing the transaction. A naive implementation of this idea would require adding  $|\text{read\_set} \cup \text{write\_set}|$  round-trips to the contention footprint to acquire the locks. Instead, the client uses an approach similar to RPC Chains [71], which cuts the round-trips required roughly in half compared to the naive approach. The client in Chardonnay sends one RPC to the first range from which it needs keys. The range acquires all the local locks, performs all the necessary local reads, and then forwards the request to the next range. The client immediately sends an RPC to the last range in the



(a) 10% Distributed Transactions.



(b) 100% Distributed Transactions.

Figure 4: Contention Microbenchmark Throughput Results

request, which holds the RPC until the request arrives. After finishing its local work, the last range replies to the client’s RPC with all the read results. When this is done, the client runs the transaction logic. If the transaction invokes a read for a key or key range (that the client already has), the client returns it immediately since it has the lock on the data (and will detect and abort the transaction if that lock is lost before commit). If transaction’s read or write set changes between the dry run and actual transaction, the client cannot serve the reads from its local cache and has to send the read requests to the ranges. We fall back to Wound-Wait for these locks.

If most transactions are likely to perform multiple read operations involving multiple network round-trips and reads from slow storage, then a developer might be tempted to parallelize those accesses, if possible, to reduce the contention footprint. Whether this is done with parallel threads or asynchronous APIs, it adds complexity to the programming model. Our scheme gets the same benefit without this complexity. On the other hand, the scheme can actually increase a transaction’s contention footprint, because lock acquisition has to be serialized. There is no overhead for the common case of transactions accessing a single range. We allow the programmer to disable ordered lock acquisition per transaction. In the future, we plan to adaptively apply the technique.

## 9 Evaluation

In this section we study how Chardonnay performs under contention (§9.1), its scalability (9.2), and its snapshot read performance (§9.3). To evaluate contention, we use a benchmark used by Calvin [76], which is inspired by TPC-C’s New-Order transaction. For scalability experiments, we use the standard TPC-C benchmark, and for read latency we use YCSB [28]. In all experiments the KV service range leaders use Standard.L8s.v2 Azure

VMs, which provide 8 vCPU cores and 64GB of memory and support accelerated networking necessary for eRPC. We place the database on directly-attached SSD for high IOPS. For the WAL, we emulate NVMe on DRAM via RAMdisk, since it is not currently offered on Azure. We advance the epoch every 10ms. All results are 10 minute averages unless otherwise stated.

### 9.1 Contention Microbenchmark

We use a benchmark introduced in Calvin [76] to evaluate Chardonnay’s performance under high contention. Each transaction in the benchmark reads 10 records, performs a constraint check on the result, and if the check passes, updates a counter in each record. The records in each KV-service range are divided into two disjoint sets: cold and hot. Each transaction can either be local or distributed. A local transaction accesses 9 records chosen at random from the cold set in the target range, and 1 record chosen at random from the hot set. A distributed transaction is similar, except it accesses 8 cold records and an additional hot record in a different range. The number of cold records in each range is much larger than available memory so cold records will be mostly served from disk. The number of hot records is determined by a parameter called the *contention index*, which is set to be the inverse of the number of hot records and represents the probability that two transactions accessing the same range will conflict. Hence, a contention index of 0.01 means that there are 100 hot records per range, while a contention index of 1 means that there is 1 hot record (which is accessed by *all* transactions touching that range). The contention index controls the degree of parallelism within each range (e.g., a contention index of 1 means that all transactions within a range are serialized).

We wrote each transaction using simple, synchronous APIs. This means that all reads are executed sequentially.

This is not a requirement, but it highlights the additional benefits of Chardonnay’s dry run and deadlock avoidance schemes, which move sequential operations outside of the contention footprint. The ordering of the reads done by each transaction is random, so there is variance in the time hot records spend under lock.

**Setup.** We use 6 ranges, and each range leader is assigned its own VM. We evaluate the following configurations of Chardonnay:

- **Baseline.** All transactions run without dry-run queries, so they do not perform prefetching or ordered lock acquisition. This is essentially a classic shared-nothing system architecture with a fast 2PC implementation, and Wound-Wait for deadlocks.
- **Prefetching-only.** Transactions run with dry-run queries, but only do prefetching and not ordered lock acquisition.
- **Chardonnay.** All transactions perform prefetching and ordered lock acquisition.

Initially, we planned to compare against CockroachDB [74] as a representative for a modern shared-nothing system. However, we realized that retrofitting the system with eRPC would be a very significant engineering effort. Running the (full SQL) system unchanged on the same experimental setup yielded low throughput (TPS per node is 90% less than Chardonnay). Hence, we use our baseline configuration for apples-to-apples comparison, as it is a good representative of the shared-nothing architecture.

We plot the throughput and abort rates under different values of contention index in Figures 4 and 5.

**Analysis.** As expected, under low contention, the dry run queries in Chardonnay are mostly wasteful and consequently the baseline configuration has slightly better throughput. Notably, full Chardonnay performs better than prefetching-only even under low contention. This is because ordered lock acquisition issues Lock & Read requests in a batched, efficient manner, as opposed to sequentially issuing an RPC per read during the transaction execution in the prefetching-only configuration. This further supports our intuition that Chardonnay’s ordered lock acquisition scheme enables writing the transactions in a simple, synchronous manner without a significant performance penalty. As contention increases, the overall throughput becomes constrained by the contention footprint, and in particular, the length of time locks on hot records also increases. The baseline configuration has the sharpest drop in throughput, since it has to issue multiple RPCs and reads from slow storage while holding locks. The full Chardonnay configuration performs best under high contention and has *zero* aborts. Prefetching-only fares much better than baseline,

even though it suffers a significant drop in throughput due to increased deadlock avoidance abort rates under contention, as well as increased contention footprint due to RPCs.

In the 10% distributed transactions case, transactions essentially never deadlock since they can only conflict on one record in the vast majority of cases. Yet, the Wound-Wait deadlock avoidance scheme is too conservative and results in many unnecessary aborts as contention increases; see Figure 5. Note that because the base configuration’s transactions have a much larger contention footprint, even a relatively modest 0.001 contention index is affected by these superfluous aborts. A less conservative scheme such as deadlock detection would not suffer from this, at the cost of taking much longer to resolve the deadlock when an actual one appears. In Chardonnay, we largely avoid deadlock aborts and only use Wound-Wait as a fall-back mechanism, as discussed in §8.

One interesting property of Chardonnay is that distributed transactions are not dramatically more expensive than local transactions. The peak throughput under low contention with 100% distributed transactions is roughly 22% lower than with only 10% distributed transactions. This makes the importance of reducing cross-partition transactions less significant, thus relieving database administrators and developers from the requirement to continually re-partition the application data to minimize cross-partition transactions [30, 35, 36, 60, 73]. The big difference in throughput between 10% and 100% ratio of distributed transactions under higher contention index values is largely because each transaction in the 100% distributed case accesses two items from the hot set, not because the transaction is distributed. This is in part because 2PC is highly optimized in Chardonnay, but also because non-distributed transactions have to go through a phase of reading the epoch before committing. Our results for the 10% distributed case show that the benefits of Chardonnay are not limited to distributed transactions.

## 9.2 Scalability

The scalability of the System R\*-style shared-nothing architecture is well established [80], but Chardonnay introduces the *read-epoch* operation during each transaction’s 2PC. Therefore, we need to ensure that the epoch service can keep up with an increasing scale.

**TPC-C New-Order.** Similar to prior work [76], we limited our TPC-C implementation to the New Order transaction, which is the bulk of the TPC-C workload including almost all distributed transactions that require high isolation. We would expect similar results if we were to run the full TPC-C benchmark. We assign each KV service range leader to a dedicated VM and have it



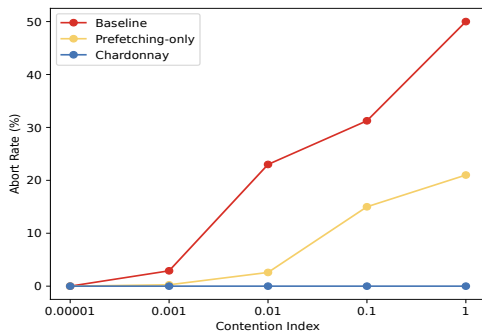


Figure 5: Abort rates for 10% distributed tx micro-benchmark. Chardonnay’s deadlock aborts are 0%.

host 10 warehouses. We limit the overall throughput to 2500 TPS per node, since we aim to evaluate the scalability of the system rather than the raw per-node peak throughput. The clients run on dedicated VMs, separate from Chardonnay nodes. (Recall that in Chardonnay, the execution of the transaction logic happens on the client.) We plot the results in Figure 6, which show stable 2PC latency as the system scales linearly.

**Comparison with Calvin.** Chardonnay is able to reach similar New-Order throughput scale as reported by Calvin [76], *without Calvin’s significant programming model restrictions* (described in §10). Calvin’s reported single-datacenter latency is much higher than Chardonnay (~100ms), but the comparison is not meaningful since it does not use fast RPC and storage. However, with 10ms epoch duration as in our setup, we expect Calvin adds 5ms to the median latency since it groups transactions into batches at the start of each epoch. Therefore, even with fast RPC and storage, we expect Calvin’s median latency to be higher than Chardonnay’s P99 latency.

**Epoch microbenchmark.** To test the limits of the Epoch service, we wrote a micro-benchmark where each thread simulates a Chardonnay client node running 2PC. We run 30 client nodes with 8 threads each, where each thread is issuing 5000 read-epoch calls per second for a total of 1,200,000 calls per second. The median latency is below 60  $\mu$ s, which is less than the median for the full Prepare phase. Since *read-epoch* runs in parallel to Prepare, this does not increase the overall 2PC latency.

### 9.3 Snapshot Read Latency

We use YCSB with 50% write and 50% read to evaluate snapshot read latency, using a setup similar to §4. Read operations run with snapshot isolation for this pur-

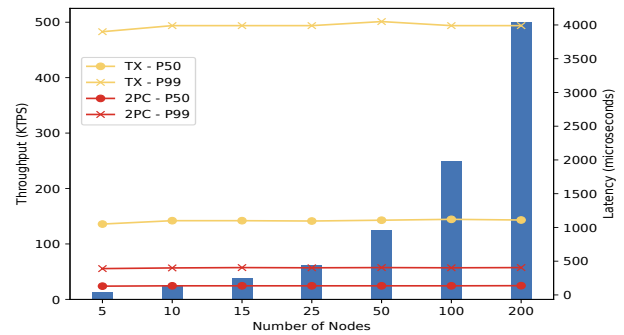


Figure 6: TPC-C New-Order transaction results.

pose. The dataset fits in DRAM since our focus is measuring protocol overhead, not IO latency. When running with a uniform distribution of keys, the median latency of reads is roughly 220  $\mu$ s. On the other hand, when running with Zipfian 0.99 distribution it increases to nearly 355  $\mu$ s. This is because most reads in the Zipfian case are going to write-hot records and hence almost always have to wait for locks to be released before they can execute. We also run the read operations with strict serializability. The median latency of the read operations increases by ~5ms since they need to wait for the epoch to advance.

### 9.4 Range Reads

We devise a simple experiment to demonstrate Chardonnay’s effectiveness for range reads. The experiment involves a single range that contains 100 records. There are two client threads, one is a writer thread that is continuously deleting and then re-inserting a random record in the range, and the other is a reader that is executing a range query to read all records. Even though the reader thread is not doing any writes, its range read query is not declared as read-only so that it runs as a read-write transaction, not as a snapshot read. We compare the number of insert operations per second in Chardonnay and the baseline from §9.1. The results are in Table 2. Without prefetching, the baseline has to execute the range read against the database each time while holding the lock on the entire range, resulting in a longer contention period and thus slowing down the writer.

## 10 Related Work

**Shared-nothing.** Spanner [29], and CockroachDB [74] are prominent modern examples of systems that utilize shared-nothing architecture, both primarily targeting inter-datacenter operations with globally-distributed workloads. Spanner uses specialized hardware to

	Chardonnay	Baseline
Insert TPS	914	197

Table 2: Range Read Results.

achieve clock synchronization guarantees that are necessary for its external consistency support. Many design choices in the system make it hard to support fast transactions within the datacenter. For instance, Spanner guarantees correctness of readers by introducing a delay for writers during the commit protocol until the clock uncertainty interval has passed, which can add many milliseconds to a transaction’s contention footprint. In contrast, Chardonnay guarantees correctness by having readers potentially wait for write locks so that the contention footprint for writers does not have to increase. CockroachDB is a system with similar emphasis on global distribution. It does not guarantee external consistency to avoid requiring specialized clocks, and instead provides the weaker single-key linearizability (which still relies on bounded clock skew). Its concurrency control protocol has optimistic components optimized for low contention.

**Shared Disk.** Shared-disk [13] is another classic DBMS architecture [25, 44] that has become popular in recent years in systems such as Amazon Aurora [78, 79], Socrates [13], and Google’s AlloyDB [2]. These systems are single-master, in which only one node actively writes the database, limiting scalability. Aurora also has a multi-master mode which does not offer serializability, and works well for partitionable workloads with little cross-partition activity. In contrast, Chardonnay can horizontally scale both reads and writes with strict serializability and supports fast cross-partition transactions.

**Shared Log.** Hyder [21] and Tango [18] scale-out compute without partitioning by utilizing a shared log that is accessed by all compute nodes. Appending to and replaying the log can be a bottleneck limiting scalability.

**Deterministic Systems.** Deterministic execution has been explored as an alternative to distributed commit in systems such as Calvin [76] and Aria [56]. A major benefit of using determinism is eliminating transaction aborts due to deadlocks [63], which Chardonnay largely achieves using its lock ordering scheme. Deterministic execution databases typically have to restrict the programming model to one-shot transactions. They also group incoming transactions into batches before executing them, which can add tens of milliseconds to latency.

Another significant limitation of most deterministic database systems is they require knowing a transaction’s read and write set upfront [75, 76]. To support dependent queries, a programmer can precede a transaction with a lower isolation *reconnaissance* query to compute

its read/write sets (e.g., OLLP [75, 76]). However, compared to dry run queries in Chardonnay, reconnaissance queries require changes to the client transaction code and run at low isolation level, exposing the code to anomalies that do not appear in the real execution. Furthermore, if the read or write set of the transaction changes between running the reconnaissance query and actual transaction, the transaction must abort. Fauna [3], a DBMS based on Calvin’s design, eliminates the need for manual reconnaissance queries at the cost of adding a round of Raft consensus to the transaction’s contention footprint, and using OCC, degrading performance under contention. Snapper [53] is a transaction library for single-node systems based on the Actor model, which enables deterministic execution for transactions that can be labeled with their read and write sets, while simultaneously supporting non-deterministic execution for transactions where this is not possible.

The Calvin paper proposes using the read set to prefetch data prior to sequencing the transaction. Prefetching in Calvin requires precisely estimating I/O latency [76]. It also happens after the reconnaissance query, adding to query latency. Notably, Calvin’s designers do not discuss range reads. Presumably even if a transaction’s entire readset is in memory, it still needs to run the range query against the database to ensure no other transaction has inserted or deleted records within that range.

**Distributed epoch-based commit.** Coco [57] is an in-memory system that applies epoch-based group commit in a distributed setting. It uses a centralized coordinator to *synchronously* commit transactions in epoch order. This requires adding many milliseconds of latency to read-write transactions. In contrast, Chardonnay is an on-disk system that guarantees the equivalence to epoch ordering, but transactions commit out of epoch order for low latency.

## 11 Conclusions

This paper presents Chardonnay, a scale-out, general-purpose, multi-versioned, on-disk transactional key-value store optimized for single datacenter deployments with fast 2PC. Chardonnay takes advantage of fast RPCs to support strictly serializable snapshot reads without relying on specialized clocks or assumptions about maximum clock skew. Chardonnay achieves high performance for high contention workloads by automatically and transparently loading and pinning data from slow storage to main memory prior to acquiring any locks, and avoids deadlocks by ordering its lock requests. We believe that the design principles of Chardonnay can also be applied in other settings, such as multi-core single-node systems for high contention workloads.

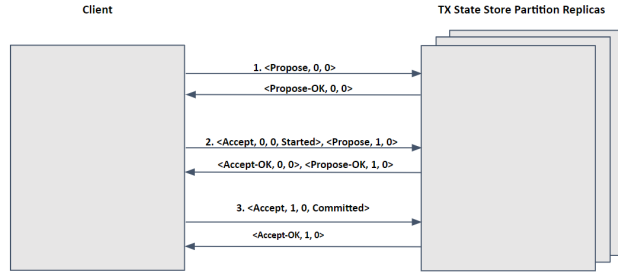


Figure 7: Recording Transaction State.

## 12 Acknowledgments

We thank our shepherd, Marc Shapiro, and the anonymous reviewers for their many useful comments and suggestions. We also thank Ryan Stutsman, Irene Zhang and Kyle Raftogianis for their feedback on earlier drafts of this work. This work was supported in part by a GE/DARPA grant, a CAIT grant, NSF awards CNS 2106530, CNS 2104292 and CNS 2143868, and gifts from JP Morgan, DiDi, and Accenture.

## A Appendix

### A.1 Optimizing 2PC

**Pipelined WAL.** Since RPCs and log writes are cheap in our system and low latency is paramount, we do not batch multiple operations into a single WAL entry. Instead, each operation (e.g., a transaction’s Prepare) has its own WAL entry (hence runs its own instance of the Paxos state-machine). Furthermore, appends to the WAL are pipelined [67] (i.e., we do not wait for the previous entries to be completely written and applied before starting a new one). Log entries are still applied to the database in log order for correctness, however. Note that a Prepare must go through the range leader, which drives appending it to the log. The long-lived leader design allows the leader to complete a log append using one RPC in the common case. Hence the latency of a Prepare operation is roughly the sum of the latencies of two RPCs and one NVMe write.

**Client-driven Commit.** As mentioned earlier, we use Paxos to replicate the state of each transaction in the transaction state store. However, to minimize the number of required round-trips, we do not designate any of the replicas as a leader. Leaders in Paxos are an optimization used in part to avoid the dueling proposers problem. Since we carefully designed the state of each transaction to be an independent Multi-Paxos log, the client is the only proposer in the vast majority of cases. So requiring it to go through a leader to run the Paxos protocol to commit (or abort) the transaction adds the latency of a su-

perfluous RPC to the Commit operation (which happens under transaction locks). Furthermore, the client utilizes a variant of the well-known technique of chaining Paxos instances together [24]. As illustrated in Figure 7, when performing the RPC to run the second (Accept) phase of Paxos to append log entry 0 (i.e., recording transaction start), the client simultaneously runs the first (Propose) phase of Paxos entry number 1 (i.e., reserving the right to propose the value of proposal number 0). Thus, it incurs the latency of only one RPC to append the decision.

### A.2 Proof Sketch of Epoch Ordering

We show that if  $e_1 < e_2$ , then  $T_1$  cannot have a dependency or anti-dependency on  $T_2$ . Given that, we can show that the transaction dependency DAG has no edges that go from a transaction with a higher to a lower epoch.

We proceed by contradiction by assuming this is false. This implies that there is (transitively) a read-write or write-write conflict between  $T_1$  and  $T_2$ , and  $T_2$  was ordered first. Therefore,  $T_2$  released a lock and sometime later  $T_1$  acquired a lock. However, since  $e_1 < e_2$ , the monotonic epoch invariant implies  $T_1$  finished execution (and acquired all its locks) before  $T_2$  did so, a contradiction as transactions do not release any locks until commit. Hence,  $T_1$  precedes  $T_2$  in the equivalent order.  $\square$

### A.3 Scaling the Epoch Service

Here we discuss briefly how to scale-out the epoch service while maintaining the correctness of our snapshot reads. The basic idea is introduce intermediary *epoch publishers* between the epoch service and its clients. Each publisher maintains a single counter (the epoch) and is Paxos replicated for high availability, much like the epoch service itself. However, the publishers do not advance the counter themselves. Instead, when the epoch is advanced by the epoch service, it issues RPCs to each publisher to advance their epoch value. The epoch service does not advance the epoch again before it updates *all* the publishers (each of which is highly available). Each client is assigned to one of the publishers, and uses the same procedure to read the epoch from that publisher exactly as it would from the epoch service itself.

This design requires slightly weakening the monotonic epoch invariant, since it is possible for a client to read a value of the epoch that is one less than the true epoch. Furthermore, when a client is assigning an epoch to a transaction, it needs to ensure the epoch is at least as high as that of any record in its read and write sets, even if the version it reads from the publisher is lower. Linearizability of snapshot reads can be ensured at the cost of additional latency, by waiting for the epoch to advance twice instead of just once.

## References

- [1] 3D Xpoint: A Breakthrough in Non-Volatile Memory Technology. <https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html>, 2023.
- [2] AlloyDB for PostgreSQL. <https://cloud.google.com/alloydb>, 2023.
- [3] Fauna. <https://fauna.com/>, 2023.
- [4] FlatBuffers. <https://google.github.io/flatbuffers/>, 2023.
- [5] gRPC. <https://grpc.io/>, 2023.
- [6] Intel® Optane™ SSD DC P5800X Series. <https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html>, 2023.
- [7] LevelDB. <https://leveldb.org/>, 2023.
- [8] RocksDB. A persistent key-value store for fast storage environments. <https://rocksdb.org/>, 2023.
- [9] Toshiba memory introduces XL-FLASH storage class memory solution. <https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html>, 2023.
- [10] Ultra-Low Latency with Samsung Z-NAND SSD. <https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low-Latency-with-Samsung-Z-NAND-SSD-0.pdf>, 2023.
- [11] yugabyteDB. <https://yugabyte.com/>, 2023.
- [12] ZippyDB: Facebook’s key value store. <https://engineering.fb.com/2021/08/06/core-data/zippydb/>, 2023.
- [13] ANTONOPOULOS, P., BUDOVSKI, A., DIACONU, C., HERNANDEZ SAENZ, A., HU, J., KODAVALLA, H., KOSSMANN, D., LINGAM, S., MINHAS, U. F., PRAKASH, N., PUROHIT, V., QU, H., RAVELLA, C. S., REISTETER, K., SHROTRI, S., TANG, D., AND WAKADE, V. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD ’19, Association for Computing Machinery, p. 1743–1756.
- [14] BACON, D. F., BALES, N., BRUNO, N., COOPER, B. F., DICKINSON, A., FIKES, A., FRASER, C., GUBAREV, A., JOSHI, M., KOGAN, E., LLOYD, A., MELNIK, S., RAO, R., SHUE, D., TAYLOR, C., VAN DER HOLST, M., AND WOODFORD, D. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD ’17, Association for Computing Machinery, p. 331–343.
- [15] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.* 7, 3 (nov 2013), 181–192.
- [16] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. HAT, not CAP: Towards highly available transactions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)* (Santa Ana Pueblo, NM, May 2013), USENIX Association.
- [17] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J. J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research (CIDR 2011)* (2011).
- [18] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP ’13, Association for Computing Machinery, p. 325–340.
- [19] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O’NEIL, E., AND O’NEIL, P. A critique of ansi sql isolation levels. SIGMOD ’95, Association for Computing Machinery, p. 1–10.
- [20] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [21] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder - a transactional record manager for shared flash. In *CIDR* (2011), www.cidrdb.org, pp. 9–20.
- [22] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The end of slow networks: It’s time for a redesign. *Proc. VLDB Endow.* 9, 7 (mar 2016), 528–539.
- [23] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP ’11, Association for Computing Machinery, p. 143–157.
- [24] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007).
- [25] CHANDRASEKARAN, S., AND BAMFORD, R. Shared cache - the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)* (2003), pp. 840–850.
- [26] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (USA, 2006), OSDI ’06, USENIX Association, p. 15.
- [27] CHEN, Y., YU, X., KOUTRIS, P., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SHU, J. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD ’22, Association for Computing Machinery, p. 19–33.
- [28] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC ’10, Association for Computing Machinery, p. 143–154.
- [29] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANKI, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.* 31, 3 (aug 2013).



- [30] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57.
- [31] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP ’07, Association for Computing Machinery, p. 205–220.
- [32] DONG, S., KRYCZKA, A., JIN, Y., AND STUMM, M. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 33–49.
- [33] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP’15)* (October 2015), ACM – Association for Computing Machinery.
- [34] EISENMAN, A., CIDON, A., PERGAMENT, E., HAIMOVICH, O., STUTSMAN, R., ALIZADEH, M., AND KATTI, S. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 65–78.
- [35] ELDEEB, T., CHEN, Z., CIDON, A., AND YANG, J. Neuroshard: Towards automatic multi-objective sharding with deep reinforcement learning. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (New York, NY, USA, 2022), aiDM ’22, Association for Computing Machinery.
- [36] ELMORE, A. J., ARORA, V., TAFT, R., PAVLO, A., AGRAWAL, D., AND EL ABBADI, A. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 299–313.
- [37] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (nov 1976), 624–633.
- [38] GRAY, J. Notes on data base operating systems. In *Advanced Course: Operating Systems* (1978).
- [39] GUO, Z., WU, K., YAN, C., AND YU, X. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD ’21, Association for Computing Machinery, p. 658–670.
- [40] GUO, Z., ZENG, X., WU, K., HWANG, W., REN, Z., YU, X., BALAKRISHNAN, M., AND BERNSTEIN, P. A. Cornus: Atomic commit for a cloud DBMS with storage disaggregation. *Proc. VLDB Endow.* 16, 2 (2022), 379–392.
- [41] HARDING, R., VAN AKEN, D., PAVLO, A., AND STONEBRAKER, M. An evaluation of distributed concurrency control. *Proc. VLDB Endow.* 10, 5 (jan 2017), 553–564.
- [42] HELLAND, P. Life beyond distributed transactions: an apostate’s opinion. In *Conference on Innovative Data Systems Research (CIDR 2007)* (2007).
- [43] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492.
- [44] JOSTEN, J. W., MOHAN, C., NARANG, I., AND TENG, J. Z. Db2’s use of the coupling facility for data sharing. *IBM Systems Journal* 36, 2 (1997), 327–351.
- [45] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI’19, USENIX Association, p. 345–359.
- [46] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 1–16.
- [47] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.
- [48] LAMPSON, B. W., AND LOMET, D. B. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1993), VLDB ’93, Morgan Kaufmann Publishers Inc., p. 630–640.
- [49] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)* (January 2015).
- [50] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. Multi-version range concurrency control in deuteronomy. *Proc. VLDB Endow.* 8, 13 (sep 2015), 2146–2157.
- [51] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD ’17, Association for Computing Machinery, p. 21–35.
- [52] LIN, Q., CHANG, P., CHEN, G., OOI, B. C., TAN, K.-L., AND WANG, Z. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD ’16, Association for Computing Machinery, p. 1659–1674.
- [53] LIU, Y., SU, L., SHAH, V., ZHOU, Y., AND VAZ SALLES, M. A. Hybrid deterministic and nondeterministic execution of transactions in actor systems. SIGMOD ’22, Association for Computing Machinery, p. 65–78.
- [54] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP ’11, Association for Computing Machinery, p. 401–416.
- [55] LOMET, D. B., AND MOKBEL, M. F. Locking key ranges with unbundled transaction services. *Proc. VLDB Endow.* 2 (2009), 265–276.
- [56] LU, Y., YU, X., CAO, L., AND MADDEN, S. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060.
- [57] LU, Y., YU, X., CAO, L., AND MADDEN, S. Epoch-based commit and replication in distributed oltp databases. *Proc. VLDB Endow.* 14, 5 (jan 2021), 743–756.
- [58] MATSUNOBU, Y., DONG, S., AND LEE, H. Myrocks: Lsm-tree database storage engine serving facebook’s social graph. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3217–3230.

- [59] MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the r\* distributed database management system. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 378–396.
- [60] QUAMAR, A., KUMAR, K. A., AND DESHPANDE, A. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), EDBT '13, p. 430–441.
- [61] RAO, J., SHEKITA, E. J., AND TATA, S. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.* 4, 4 (jan 2011), 243–254.
- [62] REN, K., FALEIRO, J. M., AND ABADI, D. J. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1583–1598.
- [63] REN, K., THOMSON, A., AND ABADI, D. J. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.* 7, 10 (jun 2014), 821–832.
- [64] ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, P. M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.* 3, 2 (jun 1978), 178–198.
- [65] ROTHNIE, J. B., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIPMAN, D. W., AND WONG, E. Introduction to a system for distributed databases (sdd-1). *ACM Trans. Database Syst.* 5, 1 (mar 1980), 1–17.
- [66] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's time for low latency. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)* (Napa, CA, May 2011), USENIX Association.
- [67] SANTOS, N., AND SCHIPER, A. Optimizing paxos with batching and pipelining. *Theoretical Computer Science* 496 (2013), 170–183. Distributed Computing and Networking (ICDCN 2012).
- [68] SERAFINI, M., TAFT, R., ELMORE, A. J., PAVLO, A., ABOULNAGA, A., AND STONEBRAKER, M. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.* 10, 4 (nov 2016), 445–456.
- [69] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 433–448.
- [70] SKEEN, D. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1981), SIGMOD '81, Association for Computing Machinery, p. 133–142.
- [71] SONG, Y. J., AGUILERA, M. K., KOTLA, R., AND MALKHI, D. Rpc chains: Efficient client-server communication in geodistributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009), USENIX.
- [72] STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering SE-5*, 3 (1979), 188–194.
- [73] TAFT, R., MANSOUR, E., SERAFINI, M., DUGGAN, J., ELMORE, A. J., ABOULNAGA, A., PAVLO, A., AND STONEBRAKER, M. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
- [74] TAFT, R., SHARIF, I., MATEI, A., VANBENSCHOTEN, N., LEWIS, J., GRIEGER, T., NIEMI, K., WOODS, A., BIRZIN, A., POSS, R., BARDEA, P., RANADE, A., DARNELL, B., GRUNEIR, B., JAFFRAY, J., ZHANG, L., AND MATTIS, P. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 1493–1509.
- [75] THOMSON, A., AND ABADI, D. J. The case for determinism in database systems. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 70–80.
- [76] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, Association for Computing Machinery, p. 1–12.
- [77] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 18–32.
- [78] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017* (2017).
- [79] VERBITSKI, A., GUPTA, A., SAHA, D., COREY, J., GUPTA, K. K., BRAHMADESAM, M., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. *Proceedings of the 2018 International Conference on Management of Data* (2018).
- [80] YANG, Z., YANG, C., HAN, F., ZHUANG, M., YANG, B., YANG, Z., CHENG, X., ZHAO, Y., SHI, W., XI, H., YU, H., LIU, B., PAN, Y., YIN, B., CHEN, J., AND XU, Q. Oceanbase: A 707 million tpmc distributed relational database system. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.
- [81] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696.
- [82] ZAMANIAN, E., SHUN, J., BINNIG, C., AND KRASKA, T. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 511–526.
- [83] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEJJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., ET AL. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 195–211.
- [84] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.
- [85] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association.
- [86] ZHONG, Y., WANG, H., WU, Y. J., CIDON, A., STUTSMAN, R., TAI, A., AND YANG, J. Bpf for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics*

in *Operating Systems* (New York, NY, USA, 2021), HotOS '21, Association for Computing Machinery, p. 128–135.

- [87] ZHOU, J., XU, M., SHRAER, A., NAMASIVAYAM, B., MILLER, A., TSCHANNEN, E., ATHERTON, S., BEAMON, A. J., SEARS, R., LEACH, J., ROSENTHAL, D., DONG, X., WILSON, W., COLLINS, B., SCHERER, D., GRIESER, A., LIU, Y., MOORE, A., MUPPANA, B., SU, X., AND YADAV, V. Foundationdb: A distributed unbundled transactional key value store. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), ACM, pp. 2653–2666.

# ScaleDB: A Scalable, Asynchronous In-Memory Database

Syed Akbar Mehdi  
The University of Texas at Austin\*

Simon Peter  
University of Washington

Deukyeon Hwang  
University of Washington

Lorenzo Alvisi  
Cornell University

## Abstract

ScaleDB is a serializable in-memory transactional database that achieves excellent scalability on multi-core machines by asynchronously updating range indexes. We find that asynchronous range index updates can significantly improve database scalability by applying updates in batches, reducing contention on critical sections. To avoid stale reads, ScaleDB uses small hash *indexlets* to hold delayed updates. We use indexlets to design ACC, an asynchronous concurrency control protocol providing serializability. With ACC, it is possible to delay range index updates without adverse performance effects on transaction execution in the common case. ACC delivers scalable serializable isolation for transactions, with high throughput and low abort rate. Evaluation on a dual-socket server with 36 cores shows that ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada on the TPC-C benchmark.

## 1 Introduction

In-memory databases [5, 10, 15, 21, 23, 42] are becoming increasingly popular: they perform well under a wide range of workloads and support requirements, such as real-time constraints, that are challenging for their disk-based counterparts [16]. They also, however, face scalability demands that sharding can only partially address: many real-world workloads have skewed access distributions [30, 47, 50, 68, 75], and the frequent hotspots they generate in individual shards require database solutions that can scale on multi-core servers.

Unfortunately, despite years of research, scaling in-memory databases on multi-core architectures remains challenging. Existing work [62, 73, 79] eliminates the bottleneck on a shared timestamp in the concurrency control protocol. Other work [37, 38, 60, 63, 77] has focused on improving the scalability of indexing structures in isolation from the database architecture. Nonetheless, current databases scale poorly on multi-core architectures (§3.1). In particular, shared range-index structures (e.g., B<sup>+</sup> trees) continue to be a main source of contention [77], and the high cost of updates to these indexes, even by unrelated transactions, is a major factor limiting scalability [62]. As fast storage via solid-state drives and persistent memory becomes the norm, contention on these structures is intensifying.

\*Currently at Google. Work done during PhD at UT Austin.

We believe that continuing to scale with these application and hardware trends requires a fresh approach. Our main observation, supported by recent work in file systems [34, 35], is that contention on shared data structures is often not fundamental, but simply an artifact of a particular system architecture. In particular, we find that contention caused by synchronous updates to sorted range-index structures is unnecessary in the common case. Our analysis (§3.2) shows that it is possible to delay many common range-index updates, without compromising on strong consistency guarantees or latency requirements for transactions. Delayed updates may be batched to reduce contention on shared range-index structures.

These observations lead us to propose a decoupled database design, centered around minimizing *unnecessary contention* among unrelated transactions. Our main technique is to decouple committing a transaction from updating the affected range indexes: we update range indexes *asynchronously*, while using scalable hash-based *indexlets* to track writes of recently committed transactions. Based on this asynchronous architecture, we design *asynchronous concurrency control* (ACC), a novel concurrency control protocol that provides serializability for concurrent transactions without compromising scalability, commit latency, or throughput. ACC is an optimistic concurrency control protocol that builds on indexlets to provide *phantomlets* for scalable phantom<sup>1</sup> detection [32]. ACC uses locks in indexlets, rather than in range indexes, to provide scalable atomic transaction commit.

We present ScaleDB, a scalable multi-core in-memory transactional database based on asynchronous concurrency control. By decoupling transaction execution from range index updates, ScaleDB can focus on improving the scalability of the former in isolation from the latter and without undesirable performance tradeoffs. By avoiding unnecessary contention on shared data structures in the common case, ScaleDB delivers scalable serializable isolation for ACID transactions, with high throughput, low commit latency, and low abort rate.

We make the following contributions:

- An analysis of the range index scalability bottleneck and of asynchronous range-index updates as a way to alleviate that bottleneck for unrelated transactions (§3).

<sup>1</sup>*Phantom* anomalies arise when insertions or deletions by other concurrently committing transactions cause two identical range scans in the same transaction to return a different set of rows.



- The design (§4) and implementation (§5) of ScaleDB, a scalable in-memory database that decouples range index management from transaction execution to allow asynchronous update of range indexes in the common case.
- Asynchronous concurrency control (ACC), a novel concurrency control protocol that provides serializability in an asynchronous database (§4.2). ACC uses *phantomlets* to scalably detect phantoms in range scans and provides scalable locks in indexlets to atomically commit transactions.
- A performance evaluation of ScaleDB on a dual-socket server with 36 cores, which shows that ScaleDB scales better than Cicada and Peloton. At scale, ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada with shared indexes on the TPC-C benchmark.

## 2 Background

Modern relational databases face challenging scalability demands. In addition to serving as backends for large-scale web applications [11, 17, 25, 31], they are offered as a service in public clouds [1, 4, 74], and must support applications that can be simultaneously write and read intensive; require both low transaction commit latency and high transactional throughput; and, increasingly, run analytical queries (on data from sources such as sensors, real-time analytics, and machine learning [19, 27]) that require maintaining a large number of indexes on every write.

In-memory databases are particularly suited to handle these diverse workload requirements, and their adoption is further facilitated by high-capacity non-volatile and disaggregated memories, as they allow for more data to be held in memory, with access latencies comparable to DRAM [53, 72]. Just as new memory technologies are shifting performance bottlenecks away from storage and towards multi-core CPU contention, such diverse workload requirements raise the bar for in-memory database scalability.

### 2.1 Prior Work

Existing efforts to improve the scalability of in-memory databases have focused on three bottlenecks: (i) range index structures; and serializable transaction isolation for both (ii) low-contention workloads and (iii) high-contention workloads (*i.e.*, transactions with dependencies). The work on range index structures has happened in isolation from the rest. This observation is key to the case for ScaleDB (§3).

**Range index structures.** Range indexes are an efficient method for data retrieval. In addition to providing exact-match lookup of database records in logarithmic time, they also allow fast scans of records in sorted order. Despite decades of work [37, 38, 48, 59–61, 63, 65], scalability of range indexes under concurrent accesses remains elusive. This is primarily due to the hierarchical nature of these data structures. For instance, in a B<sup>+</sup>tree index, inserting or deleting a new record can require modifying a chain of internal nodes all the way up

to the root. Performing such modifications atomically while supporting concurrent access from multiple threads requires synchronization [45, 77].

One approach to synchronization uses locks [48, 59, 65]. Recent optimizations [37, 38, 63] remove shared cache line contention between readers trying to acquire a lock per node, by making them optimistic. However, readers must read a version number per node to verify their optimistic assumption, which can cause contention with writers trying to increment it. Similarly, writers still contend on cache lines, trying to acquire spinlocks on individual tree nodes. Frequently accessed nodes such as the root of a B<sup>+</sup>tree or the index node at the end of the range (for append workloads) become hotspots of contention.

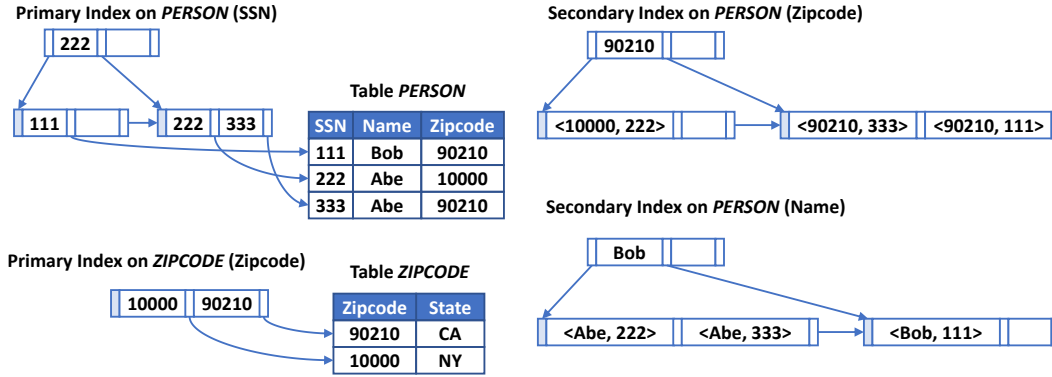
An alternative are lock-free data structures [46, 49, 61]: they use atomic operations and multi-versioning to avoid lock contention on critical sections. Yet, as recent work [45] points out, their theoretical guarantees are “mostly irrelevant to performance and scalability on multi-core hardware”, as they cannot avoid contention on global memory locations.

A recent study [77] evaluated state-of-the-art range indexes [46, 48, 60, 61, 63] on the YCSB [40] benchmark and showed that none of these indexes scale well. Even on a read-heavy workload with only 5% inserts, these indexes only scale up to 12× when increasing cores by 20×. On an insert-only workload with threads appending new inserts to the end of a range, their scalability collapses when going from a single NUMA node (20 cores) to two NUMA nodes (40 cores), with throughput dropping between 50% to 66%. The limited scalability of range indexes has been reported in previous work [62] and we expand on this analysis in §3.1.

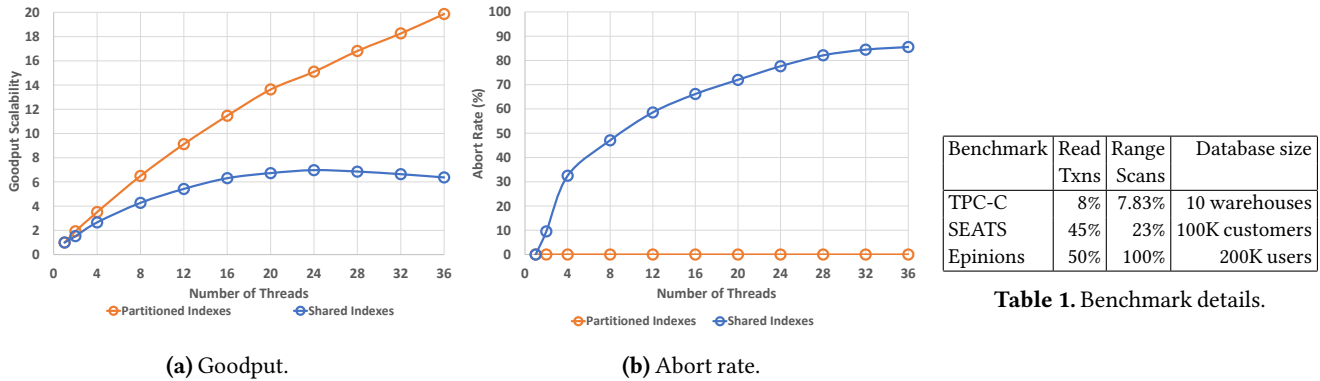
**Serializability for low-contention workloads.** The use of a shared timestamp for ordering transactions [33, 57] made timestamp allocation a principal bottleneck to the scalability of concurrency control [78]. Even when updated using atomic hardware primitives, a shared timestamp can force *unrelated transactions* to contend and results in excessive cache coherence communication. Recent work eliminates the timestamp bottleneck, but incurs high transaction commit latency due to either a high abort rate [62, 79] or batching in group commits [73].

An approach proposed by the H-store project [54, 70] avoids coordination by partitioning the database and accessing each partition from a single thread. This approach scales well for applications whose databases can be cleanly partitioned and where most transactions only access a single partition. However, many applications do not fit this profile and can experience worse performance [69].

**Serializability with dependencies.** Much work has focused on scaling serializable ordering on *contended* transactional workloads [42, 44, 51, 55, 56, 58, 62, 66, 67, 76]. Serializability requires respecting data dependencies among transactions reading and writing the same database record.



**Figure 1.** Simple database layout with range indexes. Tables are represented by primary indexes. Records are stored sorted by primary index key. Schema information is stored in a catalog (not shown). Arrows are pointers.



**Figure 2.** Cicada scalability on TPC-C ( $C_{wh=thd}$ ) with partitioned and shared indexes.

Though such dependencies are ultimately a barrier to scalability, various techniques can reduce their impact, including multi-versioning [44, 58, 62], static analysis [66, 76], exploiting commutativity in some workloads [51] and backoff [62]. These techniques are complementary to our work, which is focused on *mechanism contention*, *i.e.*, on contention that arises between unrelated transactions as an artifact of how the database implements certain mechanisms (*e.g.*, range indexes), rather than from fundamental requirements of its isolation guarantees.

### 3 The Case for ScaleDB

ScaleDB’s main contribution lies in recognizing that removing the indexing bottleneck requires looking beyond range index structures; instead, it is necessary to understand and correct the architectural design decisions that make range indexes a hotspot of contention in today’s in-memory databases.

**Range index background.** To understand the significance and structure of range indexes, consider Figure 1, which shows a simple database with two tables. Tables are implemented as collections of indexes and include one primary index and zero or more secondary indexes. For example, table PERSON

has primary index SSN and two secondary indexes, Name and Zipcode. Table records are stored on the heap and pointed to by the table’s primary index.

Range indexes have many uses. A primary range index allows quick retrieval of a table’s records by primary key for both point and range queries. Primary keys within a table must often be unique and an index can enforce this *uniqueness constraint* efficiently. Secondary indexes are also used extensively. They support analytical queries [39] and help maintain the consistency of the database by serving as *foreign keys*, *i.e.*, columns of a table that refer to a primary key of another table. For example, a foreign key constraint on the Zipcode column in the PERSON table implies that deleting the 90210 zipcode from the ZIPCODE table requires deleting all records with the 90210 zipcode from the PERSON table. The secondary index on the Zipcode column makes this operation efficient—in the Figure, the root node of the corresponding secondary tree points directly to the range of all SSNs in the 90210 zipcode; we can use these values as keys to traverse the primary index of the PERSON table.

Range indexes can limit scalability because concurrent updates to the same index, even if caused by unrelated transactions, can lead to contention. For instance, inserting or deleting a single record in a B+-tree (Figure 1) may alter its leaf node structure (a leaf node may split, or may coalesce with another leaf node), requiring the atomic update of potentially many internal nodes, all the way to the root.

### 3.1 Database Scalability Analysis

To explore the limits of database scalability, we evaluate Cicada [62], a state-of-the-art scalable in-memory database. Cicada was shown to be more scalable than several other databases [42, 55, 56, 73, 79]. However, as we show next, it still incurs range index mechanism contention.

We run TPC-C [22], a standard OLTP benchmark simulating purchase transactions on a configurable number of independent warehouses. We use a machine with two CPU sockets, each with an 18-core Intel Xeon Gold 6154 CPU. We increase the number of transaction processing server threads from 1 to 36 and use as many warehouses as threads for each data point. This configuration ( $C_{wh=thd}$ ) has very low contention, since threads (almost always) run queries on their own warehouses, thus avoiding contention on the same records with other threads. Therefore,  $C_{wh=thd}$  allows us to isolate and understand the scalability impact of mechanism contention on range indexes.

Figure 2a shows Cicada’s goodput scalability relative to a single core. Cicada stops scaling beyond 24 cores on the canonical TPC-C workload, with indexes shared between between threads. To show that scalability is limited by range index mechanism contention, we also evaluate a configuration where 8 out of the 9 TPC-C tables, and their associated indexes, are partitioned by warehouse  $id^2$ . This configuration scales well for  $C_{wh=thd}$ , but it does not generalize to more skewed workloads.

Figure 2b shows that Cicada’s poor scalability on the shared index configuration is due to an increasingly high abort rate. To reduce multi-core contention on the same index nodes by multiple threads, Cicada uses multi-version concurrency control (MVCC) for both its records and indexes; if an index node needs to be modified, Cicada creates a new version in thread local memory and installs it into the index on successful transaction commit. However, to enforce serializability, transactions that perform a range scan must, at commit time, validate that no new record matching the range predicate was inserted since the scan (*i.e.*, must avoid *phantoms*). For this purpose, at transaction commit, Cicada validates all index nodes whose key range intersected with the range scan predicate; if this validation fails, the transaction aborts. Thus, range index contention manifests in Cicada as a higher rate of transaction aborts instead of contention on index nodes.

<sup>2</sup>The default configuration of the Cicada prototype

### 3.2 When Can Range Indexes Scale?

The previous analysis demonstrates that scalability in state-of-the-art databases is primarily limited by contention on range indexes. A key contributor to this contention is that the updates to range indexes, that take place once a transaction commits, are performed *synchronously*. Of course, all indexes, whether range or hash, must, on a query, return the most recently committed record corresponding to an index key, but range indexes have an additional obligation: they must ensure that range scans issued immediately after a transaction commits will not miss any record inserted or updated by that transaction. It is to discharge this obligation that records are inserted synchronously into all primary and secondary indexes – which not only requires sorting these records with respect to *all* records already in the table, but also creates contention on the internal nodes of a range index among otherwise non-conflicting transactions.

Our design is then motivated by a simple question: *can this obligation be met without triggering a synchronous cascade of updates over shared data structures?* To move towards an answer, we run an experiment to measure the latency between the last time a record is written (inserted or updated) and when it is read as part of a range scan (*W-to-RS latency*).

We use three transactional application benchmarks (Table 1) from the OLTP-bench [43] suite, designed to evaluate modern cloud database workloads. These benchmarks range from moderately write-heavy (Epinions) to very write-heavy (TPC-C), and the percentage of read queries involving a range scan varies from a single digit to 100%. We ran these benchmarks on a MySQL 8.0 instance running on a 20 core (40 hardware threads) Intel Xeon machine, with as many clients as needed to saturate throughput. We emulate an in-memory database by setting the MySQL in-memory buffer pool to a large-enough size, so that in all three cases the entire database fits in memory and we are never disk-bound.

Figure 3a shows the cumulative distribution of the W-to-RS latency. For Epinions and Seats, we use a single curve each to characterize the behavior of all their range scans: we find that the 5th percentile W-to-RS latency is above 500ms and the median is between 8 and 85 seconds. We instead report the latency of each range scan in TPC-C separately, since they behave quite differently: *DelivSumOrderAmt*, a range scan on a primary index, responsible for 3% of all TPC-C read queries, has a median W-to-RS latency of 1ms; the other two TPC-C range scans are on secondary indexes and their median W-to-RS latencies are orders of magnitude higher. Epinions and SEATS also show lower W-to-RS latency for range scans of primary indexes, though with a much smaller ( $2\times$  to  $5\times$ ) gap. The low W-to-RS latency of *DelivSumOrderAmt* is due to the TPC-C Delivery transaction, which contains an update followed by a read on the same range in the *Orderline* table. We discuss in §4.2.3 how ScaleDB’s design avoids unnecessary aborts in such situations and therefore performs well on TPC-C (§6).

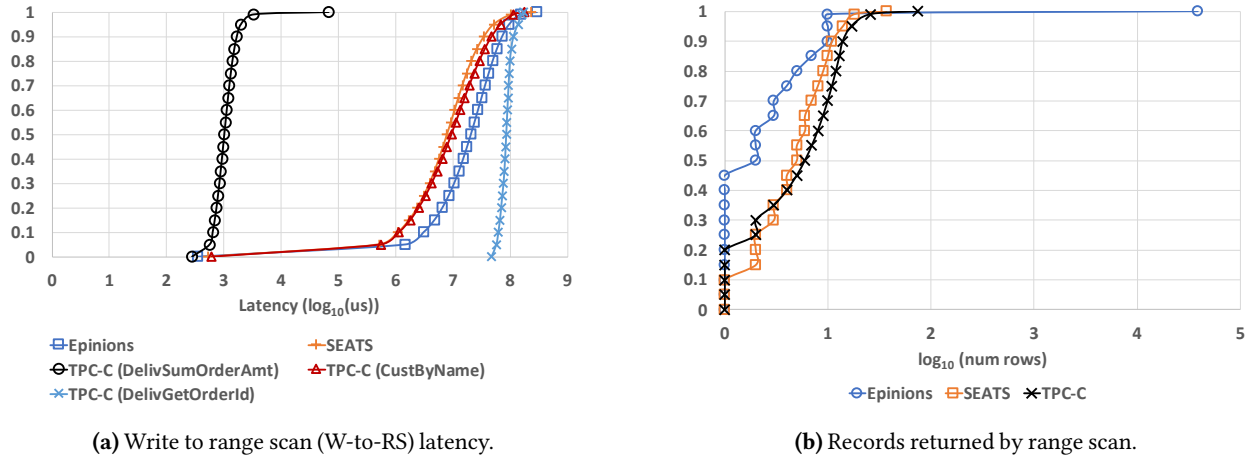


Figure 3. Range scan property distributions of three application benchmarks.

Figure 3b shows the distribution of the number of records read by range scans in each benchmark. For all benchmarks and all range scans, the median number of records read was at most 6, while the 99th percentile was at most 26 records. Epinions had two range scans in read-only transactions that read thousands of records. However these range scans comprised only 0.068% of all read queries in Epinions and had a median W-to-RS latency of at least 66 seconds.

For brevity, we omit a similar analysis for point queries, but their behavior was mixed. For instance, in TPC-C, four point queries had median Write-to-Point-Query (*W-to-PQ*) latencies ranging from  $350\mu\text{s}$  to 21ms. These point queries – from the NewOrder and Payment read-write transactions, which together comprise 90% of the benchmark – read heavily updated records in the District and Warehouse tables. At the other extreme, two point queries in TPC-C had median W-to-PQ latencies of 4 and 15 seconds.

**Conclusion.** The overall picture that emerges from this analysis is the following:

1. While point queries often read recently written records, for range queries that is the exception rather than the rule. This holds especially true for secondary indexes.
2. In the vast majority of cases, the number of records that a range query reads (especially as part of read-write transactions) is small.
3. Large range scans rarely happen and, when they do, they are usually within a read-only transaction.

These findings suggest an opportunity to fundamentally rethink how to maintain range indexes within in-memory databases. If, in the common case, synchronous updates to range indexes are not necessary to produce consistent range scans, it may be possible to design new scalable data structures that can synchronously store record updates and hold them temporarily, until they are *asynchronously* applied to the range indexes. Of course, range scans should be *always* consistent, not just in the common case, and the mechanisms

needed to enforce this guarantee should themselves be scalable. These are the opportunities and challenges that shape the design of ScaleDB.

#### Why are asynchronous range index updates scalable?

Asynchronously updating range indexes offers a host of opportunities that we seek to exploit. Accumulating a number of updates, so they can be applied as a batch to the range index, is more efficient than applying individual updates, as it avoids repeated walks of the index tree (e.g. inserts to the same B+ tree leaf node). Given the cache contention arising from concurrent walks of the range index, batched updates benefit CPU cache locality and improve performance isolation among CPU cores. They also incur less overhead for repeated lock operations, since they allow us to acquire locks only once for several updates. We can facilitate this process by sorting accumulated updates before applying them to the range index, outside of a critical section. Finally, for skewed access distributions that update the same record repeatedly within a short time span, only the last update in the batch needs to be applied to the range index, reducing the overall work required. We will see in §4.1 that asynchronous updates are scalable, while relieving the underlying range index structure of fine-grained locking, multi-versioning, and lock-free techniques. This simplifies serializability, as we will see in §4.2.

#### 4 ScaleDB Design

The foundation of ScaleDB’s design, building on the analysis in §3, is that range indexes are asynchronously updated to provide scalability. But how can this asynchronous architecture provide scalable transaction processing? And how can serializable isolation be guaranteed when range indexes are no longer kept synchronously consistent?

**Scalable transaction processing with indexlets.** To asynchronously update range indexes, we need a temporary store for writes that can be scalably maintained and flushed with



minimal overhead. We tackle this problem with a new data structure: hash-based *indexlets* that temporarily and synchronously record all range index writes. Indexlets leverage the flat structure of hash indexes to avoid contention among updates to unrelated records. A common issue with hash indexes is *rehashing* – resizing the hash index when it is at capacity [14]. Database hash indexes require rehashing, as their size cannot be known a-priori. Instead, indexlets only hold updates temporarily and are periodically merged by ScaleDB into range indexes. Thus, rehashing can be avoided by bounding the maximum number of delayed writes held in an indexlet based on the W-to-RS latency and write rate to the underlying table. We describe indexlets and how to efficiently size and scalably merge them in §4.1.

### Serializability with asynchronous range index updates.

We design *asynchronous concurrency control (ACC)*, a concurrency control protocol that provides serializability in an asynchronous database architecture. ACC integrates optimistic concurrency control (OCC) [57, 73] with asynchronous range index updates. Both are optimistic approaches: just as OCC assumes that most transactions do not contend, asynchronous range index updates assume that most W-to-RS latencies allow us to leave range indexes temporarily stale without negatively affecting goodput.

Since recent writes are held in indexlets, asynchronously enforcing serializability with good performance requires first checking indexlets on any point read, and, for range scans, efficiently detecting the small number of instances when a scan has accessed a stale portion of a range index. This check is necessary to avoid *phantoms* [32], as well as to ensure that transactions read the most recent value of each key returned by a scan.

ACC’s technique for avoiding phantoms relies on *phantom indicators*, which leverage ACC’s asynchronous design to scalably indicate the existence of a phantom to range-scanning transactions. Using the leaf nodes of the range index as partitions of its keyspace, writing transactions can produce a unique phantom indicator for each range covered by a leaf node. Each leaf node evolves through a series of version changes that happen whenever a merge to a range index affects that leaf node. Phantom indicators, uniquely derived from leaf nodes and their current version, are inserted by transactions into phantom detection indexlets (or *phantomlets*). Maintained for each range index, phantomlets allow range scanning transactions to scalably detect phantoms at commit time. We detail ACC and phantomlets in §4.2.

**Durability.** To provide durability, ScaleDB uses write-ahead redo logging. Transactions receive a globally-ordered timestamp from a system-wide clock, a hardware feature that industry trends and experimental evidence (§5.4) indicate will remain in future servers. As a result, threads can scalably log their transactions without coordination at commit time while pushing the overhead of merging logs to recovery.

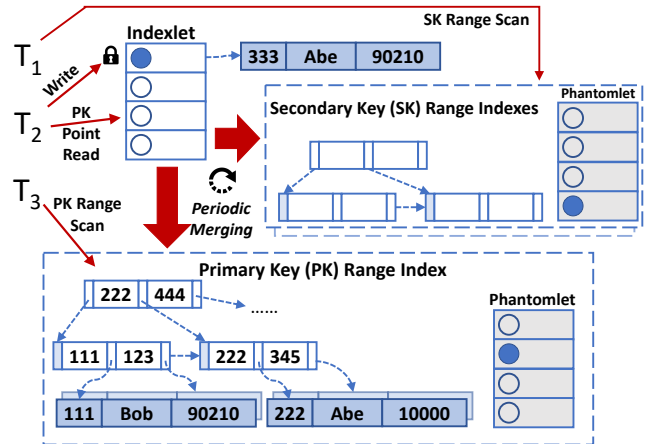


Figure 4. Asynchronous range index update for the *PERSON* table.

**Example.** To see how it all fits together, consider the example in Figure 4. Transaction  $T_1$  does a range scan by zipcode, which is executed on the appropriate secondary range index. Concurrently,  $T_2$  inserts the record with SSN 333 into the *PERSON* table and does a point read for an SSN from the same table.  $T_3$  does a range scan by SSN, which is executed on the primary range index.

Instead of synchronously updating the range indexes and potentially contending with other transactions, ScaleDB inserts  $T_2$ ’s new record, using its primary key (SSN), in the table’s indexlet and marks it as valid (filled circle). It does this atomically by acquiring a write lock on the indexlet entry. This may cause true contention if concurrent transactions access the same key, but it does not cause mechanism contention.  $T_2$  also does a point read for an SSN. To do so, it first checks the indexlet for the latest version of the record, temporarily holding a read lock on the record’s indexlet entry. It is not found there (empty circle), so  $T_2$  next reads from the primary range index. Range indexes have been read-only, and thus scalable, for this execution.

Periodically, the contents of the indexlet are merged into the underlying primary and secondary range indexes. The indexes are concurrent, so conflicting accesses by reading and merging threads are synchronized. We discuss the details of ScaleDB’s concurrent range index in §5.3. Because merging is periodic, it occurs in a coordinated and concentrated fashion when compared with synchronous range index updates.

Range-scanning transactions consult phantomlets to detect phantoms due to newly inserted records. They do this for each range index leaf node traversed as part of the range scan. To aid phantom detection, each writing transaction indicates *once* per version for a leaf node that it has inserted records. Here,  $T_2$  inserts a phantom indicator for the [222,345] leaf node into the phantomlet, indicating a possible later merge of the key with SSN 333 into that range index node. Upon a merge, not all updates might fit in the [222,345] node and the

structure of the range index might be altered during a merge. However, phantom indication is only required for unmerged records. We discard phantom indicators when the indicated records are merged. A reading transaction scanning just the [111, 123] node does not abort, as there are no phantoms indicated for this node.

#### 4.1 Asynchronous Range Index Updates

To update range indexes asynchronously, we record delayed writes in indexlets for the duration of a per-indexlet and per-thread *merge epoch*. At the end of an epoch, a thread merges its writes from the indexlet into the associated range indexes, and starts a new epoch. For a given indexlet and thread, the merge epoch ends as soon as either (i) the thread has filled a maximum *batch size* of entries in the indexlet; or (ii) a maximum epoch duration has been reached. Both batch size and maximum epoch duration are configured separately for every indexlet, and each thread decides independently for each indexlet when it has reached the end of its merge epoch.

**Indexlets.** ScaleDB uses hash-table-based indexlets with open addressing [41] to synchronously and scalably absorb concurrent, committed writes that affect range indexes. Thus, indexlets are associated with tables that have range indexes. For each such table, ScaleDB creates an indexlet, indexed by the table's primary key. If there is no primary key, ScaleDB creates an implicit primary key (a common practice [12]). The per-table indexlet naturally covers writes that affect secondary indexes, as secondary indexes refer to the primary index (as shown in Figure 1).

Recorded writes include insertions, updates, and deletions. Insertions and updates affecting a range index are simply recorded in the corresponding indexlet, and the record is updated on the heap (in per-thread arenas to avoid contention on memory allocation). Special care is required to ensure that deletes are handled consistently. Indexlets mark a record as deleted instead of deleting its key from the indexlet. This approach has two benefits: it ensures that a later read of the same key finds the deleted record in the indexlet rather than finding an older version in a range index; and it allows coalescing a key deletion followed immediately by an insert of the same key, without merging the delete into the range indexes.

**Merge epoch.** Each thread independently decides when its merge epoch ends, after which it merges the keys and record references into the table's range indexes. A thread can occupy a maximum *batch size* of  $b_i$  entries in any given indexlet  $i$  before it has to merge them into the range indexes. Too small a  $b_i$  causes contention similar to synchronous merging into range indexes. Too large a  $b_i$  results in stale range scans, which can lead to transaction aborts. We use  $b_i = \text{Expected write rate}(table_i) \times W\text{-to-RS latency}(table_i)$ .

During quiescent periods for write transactions, threads may not reach their maximum batch size quickly enough, leaving range indexes stale for too long. To avoid this, we cap the

length of the merge epoch of each indexlet separately, based on the W-to-RS latency of that indexlet's table: thus, a thread's merge epoch ends when it either reaches its maximum batch size or its maximum merge epoch length.

To make hash collisions rare, the size of indexlet  $i$  is set to  $s_i = 4 \times \#t \times b_i$ , where  $\#t$  denotes the number of threads. Given that each entry in an indexlet only occupies a single cache line, this results in modest memory consumption even for tables with a high write rate (§6.3).

**Asynchronous merging.** Each thread keeps a list of indexlet entries where it performed a write. At the end of its merge epoch, it sorts this list in primary range index key order (§3.2), and then iterates through the list, atomically merging each individual record. Merging involves updating the range index and removing the record from the indexlet, while holding a per-entry lock, thus ensuring atomicity for each key's merge. Each lock is released as soon as the key is merged into the primary index.

If secondary indexes exist, the merging thread additionally retains private copies of each record reference in thread-local storage. After the primary range index is merged, the thread then merges each secondary index, using these copies.

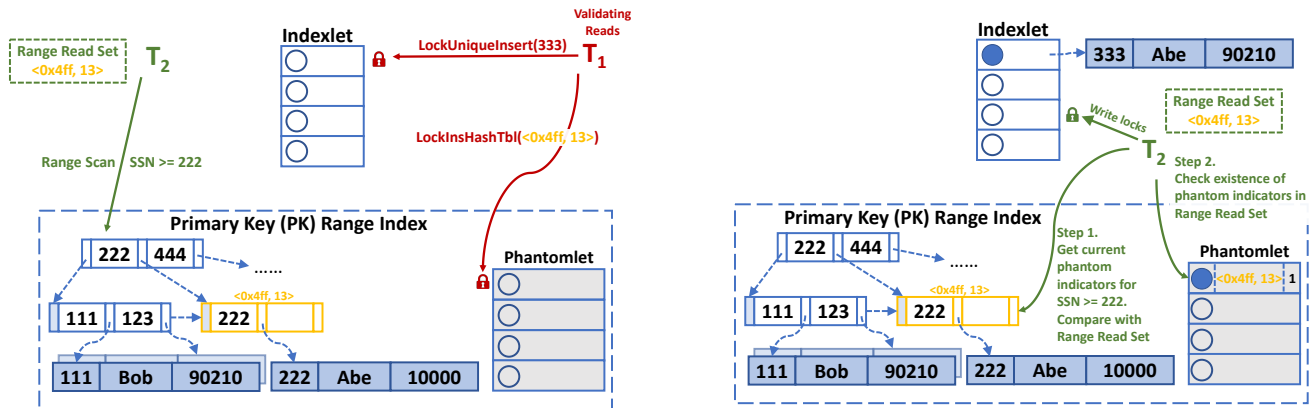
After merging each range index (primary or secondary), the merging thread also decrements any phantom indicators that it had inserted into the corresponding phantomlet during the concluded merge epoch (§4.2.1).

#### 4.2 Asynchronous Concurrency Control

We design asynchronous concurrency control (ACC), a concurrency control protocol that provides serializability within an asynchronous database. ACC is based on optimistic concurrency control (OCC), which it integrates with asynchronous range index updates. To do so, ACC uses two novel constructs: phantom indicators (§4.2.1) and locks in indexlets for atomic commit of transactional writes (§4.2.2).

**OCC** minimizes transaction contention by optimistically executing transactional reads and atomically publishing a transaction's writes at the end of its execution. To do so, OCC transactions execute in three phases—*read*, *validation*, and *commit*. During the *read* phase, reads are done optimistically, without holding locks, and are tracked in a transaction's private *read set*; writes instead are buffered in a private *write set*. The *validation* phase ensures that transactions may commit atomically. To do so, the database acquires locks on all values identified in the write set and then validates that collected values in the read set have not been altered by concurrently executing transactions. If the reads are validated, the *commit* phase commits the transaction's writes and releases its locks. Otherwise, the transaction aborts (releasing locks as well).

**ACC** extends the OCC phases and integrates them with asynchronous range index updates. During the read phase, point reads search the indexlet first, and, if they miss, search the primary range index. The same process is followed for



(a) Before validating successfully,  $T_1$  acquires locks for atomically inserting the record with SSN=333 into leaf index node [222,] and its phantom indicator <0x4ff, 13>. Concurrently,  $T_2$  does a range scan for SSN  $\geq$  222, during its read phase.

(b)  $T_2$  detects phantom indicator <0x4ff, 13> corresponding to [222,] while validating the range scan SSN  $\geq$  222. It will abort:  $T_1$  committed earlier, but  $T_2$ 's range scan missed the record with SSN=333, inserted by  $T_1$  in the indexlet.

Figure 5. Asynchronous phantom detection example.

updates and deletes, during the validation phase, allowing existing records to be brought into the primary indexlet first, before being updated in place. This guarantees that point queries always read the latest value of a record. On the other hand, range scans (from primary or secondary indexes) are executed directly on the range indexes, but need to check for phantoms at commit.

#### 4.2.1 Phantom Detection

Phantom detection is difficult in a database with asynchronously updated range indexes, as phantoms may occur in indexlets, which do not support efficient range lookup. ACC's technique for detecting phantoms leverages the leaf nodes of a range index which undergo coarse-grained version changes due to asynchronous merges by different threads. To track these changes, each leaf node  $l$  maintains a version number  $v_l$  which is incremented only when an insert or delete is merged into that node. If  $l$  splits due to an insert, then half of its keys are moved to a sibling leaf node  $m$  with  $v_m = 0$  while  $v_l$  is incremented.

To detect phantoms, ScaleDB uses a *phantomlet* per range index to perform a scalable variant of *index node validation* [73]. Phantomlets use the indexlet architecture (§4.1), but do not need merging. Inserting transactions atomically insert *phantom indicators* into phantomlets at transaction commit, indicating that they have inserted a phantom into a corresponding range index leaf node. The phantom indicator is composed of the concatenation of a leaf node  $l$ 's memory address  $M_l$  and version  $v_l$ .

At commit time, for each inserted key  $k$ , the inserting transaction asks the range index for the phantom indicator  $\langle M_l, v_l \rangle$  of the leaf node  $l$  that currently covers the range intersecting with  $k$ . If the phantom indicator does not exist in the

phantomlet, it is inserted. If the transaction validates, it atomically increments the value of the phantom indicator (initially 0). This is accomplished by locking phantom indicators as part of locking the transaction's write set (using LockInsHashTbl or LockRUDHashTbl on the phantomlets, see §4.2.2).

Threads keep track of the phantom indicators they have inserted and decrement their values at the end of their merge epoch. The last thread which decrements the value to 0, removes it from the phantomlet.

When validating a range scan, a reading transaction can use the same phantom indicator to check whether a phantom was inserted in a range covered by the leaf node *at the version it read*. To do so, ACC splits OCC's read set into two parts and extends them with additional information. For each point read, the key of a record  $r$  is stored along with a copy  $t_r^{PS}$  of the record's current commit timestamp  $t_r$  (§4.3) in a *point read set*. Storing the commit timestamp allows efficiently verifying whether the record changed, later during validation. For every range scan, ACC stores the keys of the scan results in the point read set, but also stores in a *range read set*, a phantom indicator for each range index leaf node encountered during the scan. Finally, it stores the range scan predicate in the range read set.

Read set validation happens differently for the point read set and the range read set:

- For the point read set, ACC reads from the indexlet and (if not found) searches in the primary range index. If the key of record  $r$  is not found in either index or  $t_r^{PS} \neq t_r$  ( $r$  received a write), the transaction is aborted. An optimization here is to only abort if  $t_r < t_T$ , where  $t_T$  is the timestamp allocated by this committing transaction (§4.3).
- For each range scan, ACC asks the range index for the current list  $c$  of phantom indicators that match the range scan predicate. If  $c$  is different in length than the original list  $o$

stored in the range read set, it aborts. If not, then there is still the chance that phantoms were inserted, but they have not been merged yet or they were merged but did not result in leaf node splits. ACC goes through each corresponding pair of phantom indicators in  $c$  and  $o$ , at the same index in the lists, verifying that the pair is identical, and that performing a `LockFreeRdHashTbl` (§5.2) for this phantom indicator on the phantomlet returns nothing. If any of these checks fail, it aborts.

Figure 5 shows a simple example illustrating asynchronous phantom detection.

#### 4.2.2 Atomic Commit

ACC holds locks on keys between the validation and commit phases, in order to atomically publish a transaction’s writes. Since ScaleDB writes are asynchronous, ACC locks need to cover records referenced by indexlets. Indexlets never rehash, allowing ACC to hold locks directly in indexlet entries as a way to hold locks on records.

To build transactions, ACC uses two types of locks on records: `LockUniqueInsert` is used to atomically insert a record with uniqueness constraints, while `LockUpdDel` is used to atomically update or delete an existing record. These locks are acquired on a transaction’s write set at the start of the validation phase, and released either at transaction abort or at the end of the commit phase.

**Unique insertion.** To lock for the unique atomic insert of a record, ACC performs two steps:

1. ACC searches for a duplicate record in the indexlet and, if not present, acquires a lock on an empty indexlet entry for the record to be inserted. This step is done atomically by calling `LockInsHashTbl`, provided by the indexlet. `LockInsHashTbl` acquires per-entry spinlocks along the hash probe path. If it finds an empty entry, it sets  $e_{ins}$ , the future location of the record being inserted, to that entry’s index. If the entry is not a search terminator (§5.1), it continues the search for a duplicate record, until the probe lands on a search terminator. If a duplicate is found, all spinlocks are released and the transaction is aborted. Otherwise, `LockInsHashTbl` is successful. In that case, it releases any acquired spinlocks on entries after  $e_{ins}$  in the probe path. Spinlocks on  $e_{ins}$  and entries before it in the probe path are held until `LockUniqueInsert` is released: this allows atomically inserting a record and updating search termination metadata (see §5.1) at transaction commit. With a properly sized indexlet, probe lengths are short and there is negligible mechanism contention for unique inserts.
2. ACC searches the primary range index to make sure that the key has not already been inserted there.

If either step fails, the transaction aborts. If both succeed, a lock for unique insert has been acquired. Our open addressing scheme probes indexlet entries in a deterministic order for

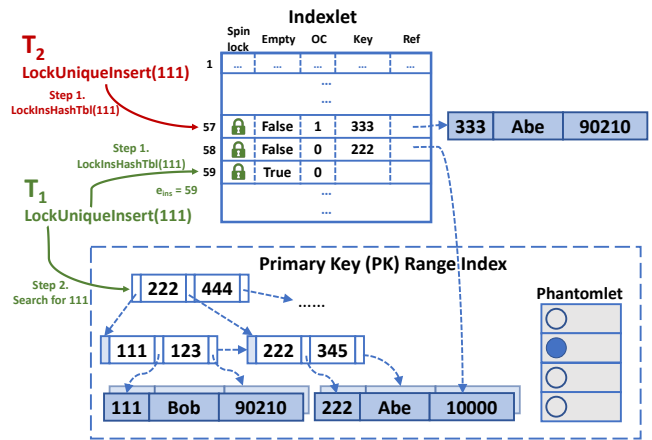


Figure 6. LockUniqueInsert Example.

each record. Hence, contending transactions attempting to insert the same record are serialized.

Figure 6 shows an example illustrating `LockUniqueInsert`. Transactions  $T_1$  and  $T_2$ , on different threads, are in their validation phase. They are concurrently trying to acquire `LockUniqueInsert` for a record with primary key (on SSN) 111.  $T_1$  acquires `LockInsHashTbl` in step 1. Its hash probe starts at entry 57 in the indexlet, which is currently occupied by a record with key 333—inserted by a recently committed transaction. Subsequently, another transaction brought the record with key 222 into the indexlet, for an update; it was inserted into entry 58 due to collision with key 333.  $T_1$ ’s hash probe acquires spinlocks along its probe path, until it lands on entry 59, which is both empty and a search terminator: thus, successfully acquiring `LockInsHashTbl` for key 111. Here, overflow counts (OC in the figure, see §5.1) are used to terminate searches (when OC = 0).

In step 2,  $T_1$  searches the primary range index for key 111, to ensure uniqueness; since it finds the record, it will abort. If  $T_1$  had been able to commit, it would have incremented the OC for entries 57 and 58 and inserted the new record (with key 111) into  $e_{ins} = 59$ , before releasing the spinlocks. Meanwhile,  $T_2$  gets serialized behind  $T_1$  (on entry 57’s spinlock), trying to acquire `LockInsHashTbl`. It will eventually abort as well.

**Update and deletion.** To acquire a `LockUpdDel`, ACC performs two steps:

1. It searches the indexlet for the record and, if found, locks the entry. This step is done atomically by calling `LockRUDHashTbl`, provided by the indexlet. `LockRUDHashTbl` is simpler than `LockInsHashTbl`, since it does not need to atomically enforce uniqueness or maintain the metadata for search termination. It acquires per-entry spinlocks along the hash probe path, but releases each spinlock as it moves to lock the next entry in the path. A probe can end when it either finds the record or lands on a search terminator entry.



In addition to its use in the first step of LockUpdDel, LockRUDHashTbl is also used to atomically search the indexlet for point queries, during the read phase of the transaction.

2. If the record was not found in the indexlet, ACC acquires LockInsHashTbl for the record, fetches the record from the range index, inserts a reference to the record in  $e_{ins}$  and then downgrades the lock to LockRUDHashTbl, which involves releasing the spinlocks on the entries before  $e_{ins}$  in the probe path.

For range updates or deletes, we search the range indexes directly and acquire LockUpdDel for every key satisfying the predicate. If there is not enough space in the indexlet, the transaction aborts. In this rare case, the indexlet is merged and temporarily disabled to retry the transaction synchronously, re-enabling the indexlet after the transaction commits.

### 4.2.3 Repairing Stale Range Scans

During the read phase, ACC can repair stale scans before returning them, to reduce the chance of a later transaction abort. This is typically done for scans used in a later update or delete query. For instance, the TPC-C Delivery transaction has a range scan that returns the earliest order within a district in the NEW-ORDER table and then deletes that order in the next query. This transaction can abort, even for a single thread, if the scan is done on the range index, but the earliest order returned by the scan has already been marked deleted in the indexlet in a previous Delivery transaction.

ACC repairs such scans, prior to returning them, by looking up each key in the indexlet to check if it has been updated or deleted. If so, it repairs the scan to return the latest version. To avoid paying this cost for all range scans, the client can explicitly set this option in the query for scans that will be updated or deleted.

ACC also maintains a per-thread per-table index of the keys which were inserted by each thread during its current merge epoch. When returning a range scan, ACC repairs it by merging any records returned by running the same scan on the local index as well. This avoids spurious aborts by the phantom detection algorithm (§4.2.1), due to keys that were inserted by the same thread in a prior transaction and are waiting to be merged into the range indexes.

### 4.3 Durability

ScaleDB achieves durability using write-ahead logging to a redo log. Each worker thread writes to its own separate log, without coordinating with any other worker thread. To ensure that transactions do not read values that have not been made durable, a thread only releases write locks and replies back to the client once it has logged the transaction to its redo log. Each redo log entry contains the new values of the keys written by the transaction  $T$  as well as a commit timestamp  $t_T$  assigned to it during the validation phase, after all the locks have been acquired by ACC. This timestamp, unique for each transaction, is derived from a scalable system-wide clock

(§5.4) and is consistent with  $T$ 's place in the serializable order. During recovery, ScaleDB first merges all the per-thread transaction logs in timestamp order, and then replays them.

To see why ScaleDB is recoverable despite uncoordinated logging, consider the example of three transactions  $T_1 \xrightarrow{ww} T_2 \xrightarrow{rw} T_3$ , each of them running on a separate thread.  $T_1$  writes  $x_1 = 42$  and  $T_2$  read-modify-writes that value to  $x_2 = 52$ , thus creating both a write-after-write (ww) and read-after-write (wr) dependency with  $T_1$ . Next,  $T_2$  reads  $y_1 = 33$ ; later  $T_3$  read-modify-writes it to  $y_2 = 36$ , creating a write-after-read (rw) dependency between  $T_2$  and  $T_3$ .

Because ScaleDB only releases write locks after the log entry has been made durable, if  $T_1$  is not logged, then  $T_2$  will either read  $x_0$  or it will wait for  $T_1$ 's write lock to be released to read  $x_1$ . Thus, after a crash, if  $T_2$  read  $x_1$  and is logged, then  $T_1$  must be logged as well. This argument extends transitively to a chain of such direct dependencies.

The second possibility is that after a crash  $T_2$  is not logged, but both  $T_1$  and  $T_3$  are. In this case, ScaleDB must not have committed  $T_2$  and replied back to the client. Thus, it will recover only  $T_1$  and  $T_3$ , in order, which is fine. Notice that, if, in fact  $T_2$  *does* get successfully logged, ScaleDB's system-wide timestamps allow correctly ordering  $T_2$  and  $T_3$ 's log entries at recovery, despite the fact that there was no direct communication among them.

### 4.4 Correctness

Using ACC, ScaleDB guarantees serializability [28], with the additional guarantee that the equivalent serial order is one where transactions are ordered by their commit timestamps. ACC derives its correctness guarantees in part from the guarantees provided by the locks and data structures it builds upon, as well as its descendance from OCC, which guarantees serializability [73]. The key difference from OCC is that ACC must deal with ScaleDB's asynchronous updates to range indexes. Our proof of correctness [64] shows that ACC's atomic commit and phantom detection protocols provide serializability in this scenario.

## 5 Implementation

We implement ScaleDB by modifying the Peloton [18] in-memory SQL database, written in C++. We replace the storage back-end, while retaining the code for networking, SQL parsing, query planning and query optimization.

### 5.1 Indexlet and Phantomlet Hash Table

Our indexlet and phantomlet implementations build on a simple open-addressing [41] hash table which uses linear probing for resolving collisions. We considered more sophisticated open-addressing schemes like Cuckoo hashing [2, 7] but found that the ability to hold transactional locks would have been complicated by displacement of keys and the fact that the cuckoo hashing probe path is an undirected graph with a possible cycle, which could have caused deadlocks.

Also recall that our hash table does not need to rehash: thus we can avoid mechanism contention on maintaining a count of occupied entries in the entire hash table.

One issue with using an open addressing hash table is how to ensure that searches terminate correctly after merging. When removing a record  $r$  from an indexlet entry, we cannot simply mark the entry as empty, because then any records displaced by  $r$  would not be found on a subsequent lookup (the search would terminate at  $r$ ). Tombstones, which are traditionally used in open addressing tables, have the problem of accumulating and making search probes ever longer. Instead, we used a scheme used by the recent non-concurrent F14 hash table [29, 36]. Each entry maintains an *overflow count*, that is incremented whenever an insert probe finds the entry already occupied. When removing a record reference at the end of a merge epoch, we atomically decrement the overflow counts on its probe path, before marking it as free. Similarly, when inserting a record reference, we atomically increment the overflow counts on its probe path. An entry whose overflow count is zero is a search terminator (§4.2.2).

## 5.2 Lock-Free Reads

To avoid reader contention on the same indexlet or phantomlet entries, ScaleDB provides *LockFreeRdHashTbl* (based on seqlocks [3]). To implement these, we add a version number to the per-entry spinlocks in the indexlet or phantomlet hash tables. Writers (doing inserts, updates or deletes) increment the version number after acquiring the spinlock but before any writes. At spinlock release, the version number is incremented again. Readers do not acquire the spinlocks but instead read the version number, before and after they perform the read. If the version number changed during the read or it is initially odd in value, then there was interference from a concurrent writer and the reader retries.

The limitation of this design is that it cannot be used if the data being read has internal pointers; otherwise, writers could invalidate pointers that a reader had already followed. To solve this, we can use Read-Copy-Update (RCU) [24] for implementing *LockFreeRdHashTbl* [73]. However, RCU can add significant complexity to the design; e.g., it requires garbage collection of previous versions of the data, after ensuring that no readers are actively reading it.

Our current prototype does not implement RCU. Instead, we only use *LockFreeRdHashTbl* for use-cases where the data does not have internal pointers; e.g., during the ACC validation phase, we use it to atomically search phantomlets, thus avoiding mechanism coordination between threads searching for phantom indicators for the same leaf node version of a range index. We also use *LockFreeRdHashTbl* to validate point reads in indexlets with fixed-length keys. If the data has internal pointers, we instead use *LockRUDHashTbl* (§4.2.2).

## 5.3 Concurrent Range Index

Our range index implementation is a B+ tree with optimistic latch coupling (OLC) [60], used in a recent study [77] that compared the scalability of state-of-the-art range indexes. In the OLC tree, reads do not acquire the per-node spinlocks when traversing the tree. Instead, they validate a per-node version number by reading it before and after reading the node's contents. If the two versions are not the same, they restart their traversal. Writers initially traverse like readers, but restart and acquire spinlocks along the path if they detect interference from another writer or if nodes need modification.

## 5.4 System-wide Synchronized Clock

For scalable durability, ScaleDB assigns timestamps to transactions derived from a system-wide synchronized clock. Synchronized hardware clocks are available on modern multi-core processors, such as the timestamp counter (TSC) on recent Intel x86 processors, which runs at a constant rate. Intel has indicated [52] that “*this is the architectural behavior moving forward*” and that “*the OS may use invariant TSC for wall clock timer service*”. As a result Linux uses the TSC as the clock source on x86 across multiple CPU sockets, after running boot-time tests to ensure synchronization [8, 9]. Recent work [34, 35] on multi-core filesystems has used it for scalable ordering across cores. Finally, virtual machines also provide synchronized virtual TSCs by either using the underlying hardware (fast) or emulating it if not synchronized (slow) and even across migrations [20, 26].

On architectures where a system-wide TSC is not available, it is possible to use a dedicated timing thread [71] for handing out timestamps. This approach requires a core dedicated to the timing thread, which continuously increments a local variable and then stores the value to a global time variable. A thread requiring a global timestamp simply reads the time variable. On the Intel Skylake architecture, such a timing thread increments the local variable every 0.87 cycles which is actually 15% faster than the TSC [71], but requires a core.

## 6 Evaluation

Our evaluation aims to understand how ScaleDB performs in terms of throughput scalability of committed transactions on various workloads, including YCSB and TPC-C, and how the various ideas in the design of ScaleDB contribute to performance. Our comparison baselines are Peloton, upon which ScaleDB is built, and Cicada.

Our evaluation answers the following questions:

1. What is the query scalability of ScaleDB when compared to Peloton (§6.1)? We use YCSB to answer this question.
2. How does ScaleDB scalability compare to Cicada when guaranteeing serializability for transactions (§6.2)? Is the transaction abort rate affected? We evaluate TPC-C.

3. Is the ScaleDB asynchronous architecture a scalable design (§6.3)? We evaluate the scalability of indexlets (phantomlets) and system-wide timestamps given that these mechanisms are necessary for a scalable, asynchronous database.

**Testbed.** All machines in the evaluation have 2×18-core Intel Xeon Gold 6154 CPUs with 36 cores. 192GB of memory is divided across two NUMA nodes. Each machine has a Mellanox ConnectX-5 NIC, operating at 100Gb/s. For networked benchmarks, we run a single database server and 4 client machines. Each client machine runs as many processes of the OLTP benchmark suite [13] as needed to saturate the database server. Accordingly, all experiments report peak throughput.

### 6.1 Asynchronous Index Update

We evaluate ScaleDB’s scalability of asynchronous updates to a single range index and compare to Peloton. For this purpose, we use the Yahoo! Cloud Serving Benchmark (YCSB) [40] read-insert workload. To generate enough load, we access the database from 4 networked YCSB benchmark client machines. The YCSB benchmark defines a single table with an integer primary key and 10 string columns, each of size 100 bytes. Peloton uses the lock-free Bw-Tree [77] as the underlying primary range index on the integer key.

All experiments use 36 server threads, with each thread pinned to a separate core. We show scalability by increasing the number of client terminals sending operations to the database server. For ScaleDB, we set the maximum merge epoch duration to 100 ms and the maximum batch size per thread to 1,000 entries. Prior to running each experiment, we load the table with 1 million records. We use a Zipfian distribution for reads with  $\theta = 0.99$  to simulate a skewed workload. For inserts, each client thread adds new records sequentially within its own interval of the primary key space, starting after the already inserted 1 million records, to avoid uniqueness conflicts.

**Mechanism contention.** Figure 7a shows terminal scalability for two points of read-insert intensity. The read-insert workload has only mechanism contention—reads and inserts are to disjoint keys. For 95% reads, both ScaleDB and Peloton scale with similar performance until all server cores are saturated. This is not surprising. Peloton’s range index scales well when a workload is read-intensive. For a write-intensive workload with 50% inserts, Peloton’s throughput collapses, while ScaleDB maintains 9.5× Peloton’s throughput at scale.

To detail this effect, we examine the sensitivity of both systems to increasing write intensity by varying the fraction of inserts in the workload, fixing the number of terminals to 160. Figure 7b shows that Peloton’s throughput quickly collapses with increasing write intensity (knee-point at 20% inserts), while ScaleDB’s throughput gradually declines. ScaleDB loses 46% of its peak throughput when the workload is write-only.

### 6.2 Serializability

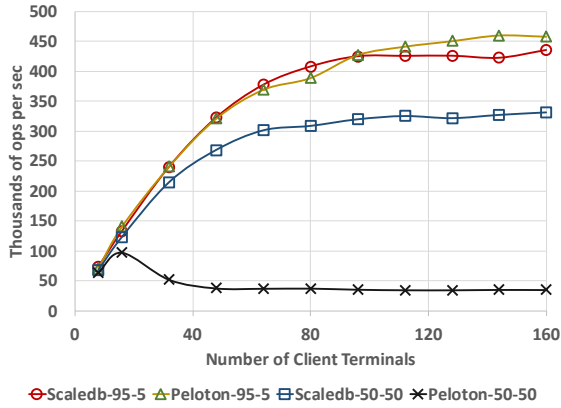
We now evaluate asynchronous scalability with serializable transactions on the TPC-C benchmark, which has multiple tables and several primary and secondary range indexes. We compare with the Cicada [62] database. Cicada’s prototype does not have a network layer and it uses a TPC-C implementation linked with the database binary, calling directly into the Cicada function call API as opposed to sending SQL calls across the network. For a fairer comparison, we do the same for ScaleDB. Cicada’s prototype also pre-allocates all of its memory using huge pages. Recent work from Huang et al. [51] has recommended avoiding this strategy since it “changes system dynamics significantly—for instance, pre-allocated indexes never change size”. Further, Huang et al. show that Cicada, with pre-allocation, experiences a performance collapse at high core counts due to memory exhaustion, which we observed as well. Therefore, we modify Cicada to instead use jemalloc [6], which is what ScaleDB uses for memory allocation. Finally, Cicada simplifies multi-column keys by reducing them to 64-bit integers (using assumptions about the maximum range of each column). Thus, all key comparisons in Cicada are between single 64-bit integers, while ScaleDB stores and compares multi-column keys (with possibly varying column types). Hence, the baseline performance in this evaluation is biased against ScaleDB. We report self-normalized scalability, in addition to raw transactional throughput (Figure 8), for a more complete picture.

TPC-C does not run range scans on the WAREHOUSE, DISTRICT and ITEM tables. Cicada uses hash indexes for these tables and we do the same for ScaleDB. For the other tables, ScaleDB’s maximum per-thread batch sizes are calculated using the method outlined in §4.1. The New-Order and Delivery transactions exert a very small W-to-RS latency on the NEW-ORDER table, requiring this table’s maximum batch size to be set to 0 (*i.e.*, synchronous merging into the range index). The remaining tables have a batch size of 2,048.

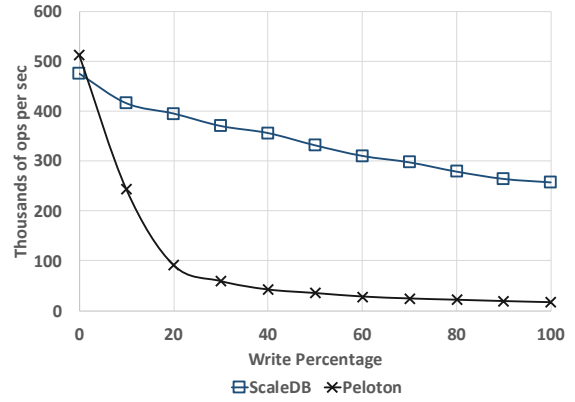
Figure 8 shows the TPC-C evaluation. The setup for these experiments is the same as that of Figure 2. ScaleDB does not have a partitioned index configuration, so all results for ScaleDB use shared indexes. On the canonical TPC-C benchmark (Figure 8a), ScaleDB scales 22.3× on 36 cores (relative to its single core throughput), which is significantly better than Cicada’s scalability (with shared indexes) of 6.4×. At scale, ScaleDB’s raw throughput is 1.8× higher than Cicada.

Partitioned indexes show the upper bound for Cicada’s scalability. ScaleDB, with shared indexes, achieves better self-normalized scalability than Cicada with partitioned indexes (Cicada scales only 20× over 36 cores). At scale, ScaleDB’s raw throughput is 60% of Cicada. Of course, Cicada’s partitioned indexes do not generalize to skewed workloads.

We also evaluate a workload (NewOrd-Deliv, Figure 8b) consisting of TPC-C transactions New-Order and Delivery in equal proportions. On this more index-contended benchmark,

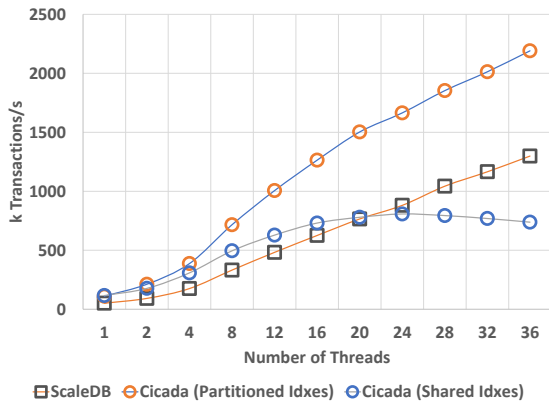


(a) Throughput.

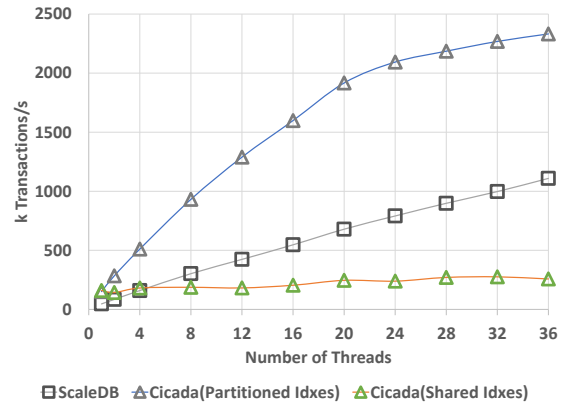


(b) Write sensitivity.

Figure 7. YCSB read-insert workload. 95-5 is 95% reads and 5% inserts. 50-50 is 50% reads and 50% inserts.

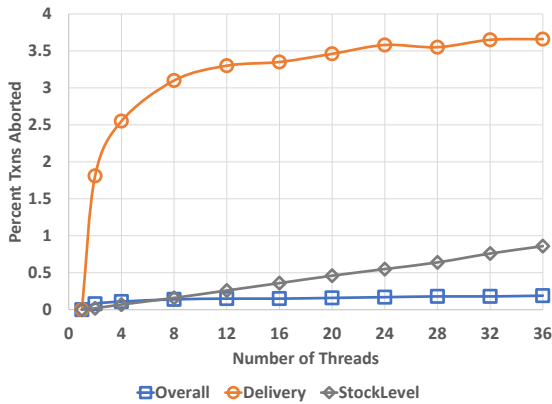


(a) TPC-C.

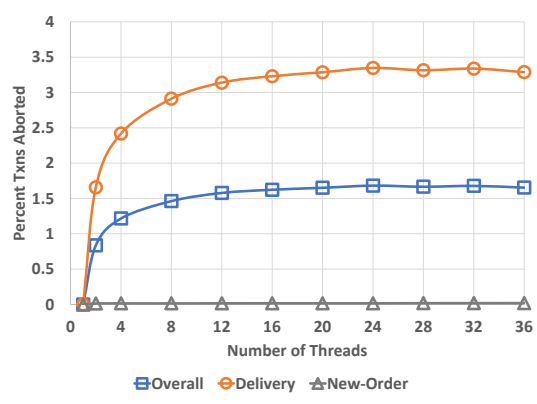


(b) NewOrd-Deliv.

Figure 8. ScaleDB vs Cicada goodput scalability on the TPC-C benchmark. Goodput counts only committed transactions.



(a) TPC-C.



(b) NewOrd-Deliv.

Figure 9. ScaleDB abort rate.



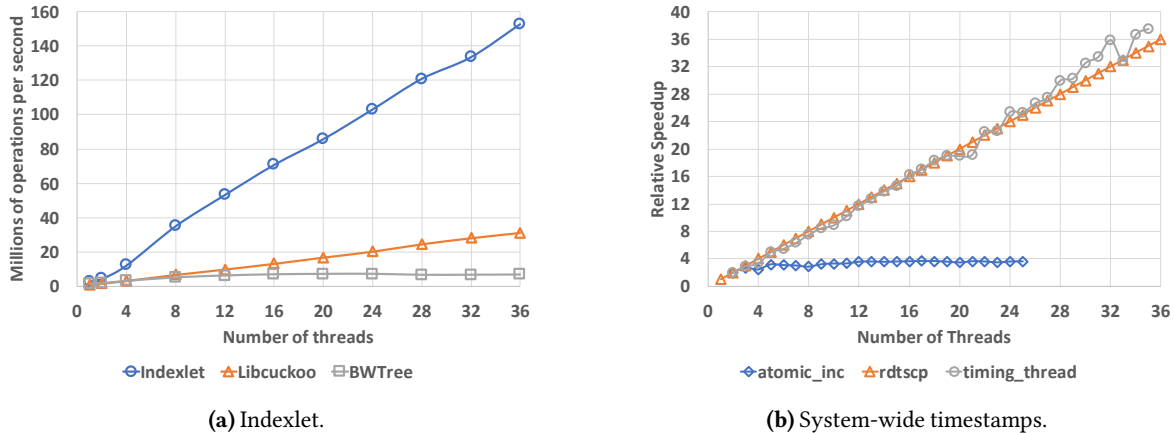


Figure 10. ScaleDB mechanism scalability.

ScaleDB maintains its scalability, while Cicada’s scalability is severely impacted. ScaleDB scales 24× over 36 cores, compared to only 1.6× for Cicada with shared indexes. ScaleDB’s throughput is 4.3× higher than Cicada. With partitioned indexes, Cicada scales 15.5×, still worse than ScaleDB using shared indexes. At scale, ScaleDB (with shared indexes) provides 48% of Cicada’s throughput (with partitioned indexes).

Given ScaleDB’s asynchronous design and the fact that transactions can do stale reads from range indexes in between batch merges, an important concern is how the abort rate behaves with an increasing number of threads. Figures 9a and 9b show this evaluation. On the canonical TPC-C benchmark, only the Delivery and StockLevel transactions have a non-negligible abort rate. The Delivery abort rate stabilizes at 3.5% around 20 cores (for both workloads), which implies that ScaleDB continues scaling even beyond 36 cores. The Stock-Level abort rate stays under 1%, even for 36 cores.

We also evaluated sensitivity of the abort rate to batch size, but found that our workloads were not very sensitive to even significant variations around the initial batch size—calculated according to the expected write rate for the corresponding table (§4.1). Accordingly, we omit those results for brevity.

### 6.3 ScaleDB Mechanisms

**Indexlets.** We evaluate the scalability of indexlets against libcuckoo [7], an optimized concurrent hash table, and the BwTree [61, 77], a recent, lock-free range index structure. This evaluation is performed on a microbenchmark (included with libcuckoo) with a 50% read and 50% insert workload consisting of 64-bit integer keys and values. As Figure 10a shows, indexlets achieve nearly 5×libcuckoo and 25×BwTree throughput at 36 cores. Open addressing in indexlets provides better scalability than cuckoo hashing and the flat structure of hash tables scales better than tree indexes.

**Memory Overhead.** Indexlets have low memory overhead. Each indexlet entry only contains the primary key, a reference to the actual database row, and a small amount of meta-data (e.g., a spinlock). For primary keys composed of integer columns, such as those in TPC-C tables, an indexlet entry can fit within a cache line (i.e. 64 bytes). As a result, the maximum size of an indexlet in our benchmarks was ~60MB, even for tables (e.g. the TPC-C Orderline table) which absorbed millions of record inserts per second at peak. For phantomlets, the memory overhead is even more modest, as their entry count is sized according to the expected number of leaf index nodes used for inserts per epoch. Accordingly, the maximum size of phantomlets in our benchmarks was lower than 1MB.

**System-wide timestamps.** We evaluate the TSC and timing thread approach (§5) and compare with an atomic increment as a global timestamp. As Figure 10b shows, both timing thread and TSC approaches scale linearly to 36 cores, while the atomic increment does not scale beyond 4 cores.

## 7 Conclusion

ScaleDB is an asynchronous in-memory database that provides scalability and serializability for ACID transactions. ScaleDB asynchronously updates range indexes by temporarily holding writes in indexlets that are merged periodically into range indexes. ScaleDB uses asynchronous consistency control (ACC) to provide transaction serializability. ACC extends OCC with asynchronous phantom detection via phantomlets and atomic transaction commit using locks in indexlets, rather than range indexes. For durability, ScaleDB uses system-wide time stamp counters for scalable redo logging. ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada on the TPC-C benchmark.

**Acknowledgments.** We thank the anonymous reviewers and our shepherd, Murat Demirbas, for their feedback. This work was supported by NSF grant 2227066.

## References

- [1] Azure SQL database: Managed, intelligent SQL in the cloud. <https://azure.microsoft.com/en-us/services/sql-database/>.
- [2] Cuckoo Hashing. <https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf>.
- [3] Driver porting: mutual exclusion with seqlocks. <https://lwn.net/Articles/22818>.
- [4] Google Cloud Spanner. <https://cloud.google.com/spanner/>.
- [5] HyPer – A Hybrid OLTP&OLAP High Performance DBMS. <https://hyper-db.de/>.
- [6] jemalloc. <https://jemalloc.net/>.
- [7] libcuckoo. <https://github.com/efficient/libcuckoo>.
- [8] Linux TSC Cross Socket Reliability. <https://github.com/torvalds/linux/blob/c2131f7e73c9e9365613e323d65c7b9e5b910f56/arch/x86/kernel/cpu/intel.c#L249>.
- [9] Linux TSC Synchronization. [https://github.com/torvalds/linux/blob/master/arch/x86/kernel/tsc\\_sync.c](https://github.com/torvalds/linux/blob/master/arch/x86/kernel/tsc_sync.c).
- [10] MemSQL. <https://www.memsql.com/>.
- [11] MyRocks: A space- and write-optimized MySQL database. <https://engineering.fb.com/core-data/myrocks-a-space-and-write-optimized-mysql-database/>.
- [12] MySQL 8.0 Reference Manual: Clustered and Secondary Indexes. <https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html>.
- [13] OLTP-Bench. <https://github.com/oltpbenchmark/oltpbench>.
- [14] Resizing Hash Tables. <https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf>.
- [15] SAP HANA. <https://www.sap.com/products/hana.html>.
- [16] The Forrester Wave™: In-Memory Databases, Q1 2017. <http://www.oracle.com/us/corporate/analystreports/forrester-imdb-wave-2017-3616348.pdf>.
- [17] The Infrastructure Behind Twitter: Scale. [https://blog.twitter.com/engineering/en\\_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html](https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html).
- [18] The Peloton self-driving SQL database management system. <https://github.com/cmu-db/peloton>.
- [19] Time-series data: Why (and how) to use a relational database instead of NoSQL. <https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/>.
- [20] Timekeeping in VMware Virtual Machines. <https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf>.
- [21] TimesTen: Fastest OLTP database, ultra high availability, elastic scalability. <https://www.oracle.com/database/technologies/related/timesten.html>.
- [22] TPC-C Benchmark. [http://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-c\\_v5.11.0.pdf](http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf).
- [23] VoltDB. <https://www.voltdb.com/>.
- [24] What is RCU, Fundamentally? <https://lwn.net/Articles/262464/>.
- [25] Why Uber Engineering Switched from Postgres to MySQL. <https://www.uber.com/blog/postgres-to-mysql-migration/>.
- [26] Xen TSC (time stamp counter) and timekeeping discussion. <http://xenbits.xen.org/docs/4.13-testing/man/xen-tscmode.7.html>.
- [27] Firas Abuzaid, Peter Bailis, Jialin Ding, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: Prioritizing attention in fast data. *ACM Trans. Database Syst.*, 43(4):15:1–15:45, December 2018.
- [28] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, 1999.
- [29] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 01 1974.
- [30] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.
- [31] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, 2017.
- [32] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.
- [33] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.
- [34] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, 2017.
- [35] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT, September 2014.
- [36] Nathan Bronson and Xiao Shi. Open-sourcing F14 for faster, more memory-efficient hash tables. <https://engineering.fb.com/developer-tools/f14/>.
- [37] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, pages 257–268, 2010.
- [38] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoon Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 181–190, 2001.
- [39] Biswapesh Chattopadhyay, Sagar Mittal, Roei Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Priyam Dutta, Selcuk Aya, Vera Lychagina, Brett Elliott, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, and David Lomax. Procella: unifying serving and analytical data at YouTube. *Proceedings of the VLDB Endowment*, 12:2022–2034, August 2019.
- [40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.
- [41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, 3rd edition, 2009.
- [42] Cristian Dacuon, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, 2013.
- [43] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.
- [44] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.
- [45] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, pages 9–21, 2017.
- [46] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, 2004.
- [47] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the 13th EuroSys*

- Conference, EuroSys '18, 2018.
- [48] Goetz Graefe. A survey of B-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010.
- [49] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity, SIROCCO'07*, pages 124–138, 2007.
- [50] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII*, pages 1–7, 2014.
- [51] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.*, 13(5):629–642, January 2020.
- [52] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B, chapter 17, pages 17–41. November 2018.
- [53] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. <http://arxiv.org/abs/1903.05714>, 2019.
- [54] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. Hstore. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.
- [55] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 1675–1687, 2016.
- [56] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 691–706, 2015.
- [57] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.
- [58] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.
- [59] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.
- [60] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN '16*, pages 3:1–3:8, 2016.
- [61] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering, ICDE '13*, pages 302–313, 2013.
- [62] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17*, pages 21–35, 2017.
- [63] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, pages 183–196, 2012.
- [64] Syed Akbar Mehdi. *Scalability through Asynchrony in Transactional Storage Systems*. PhD thesis, The University of Texas at Austin, 2022. Appendix 2.
- [65] C. Mohan and Frank Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, SIGMOD '92*, pages 371–380, 1992.
- [66] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the 14th EuroSys Conference 2019, EuroSys '19*, 2019.
- [67] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 677–689, 2015.
- [68] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. An analysis of load imbalance in scale-out data serving. *SIGMETRICS Perform. Eval. Rev.*, 44(1):367–368, June 2016.
- [69] Andrew Pavlo. What are we doing with our lives? Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17*, page 3, 2017.
- [70] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 61–72, 2012.
- [71] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks, 2017. <https://arxiv.org/abs/1702.08719>.
- [72] Debendra Das Sharma. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects, HOTI '22*, pages 5–12, 2022.
- [73] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13*, pages 18–32, 2013.
- [74] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 789–796, 2018.
- [75] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. Fat caches for scale-out servers. *IEEE Micro*, 37(2):90–103, March 2017.
- [76] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 1643–1658, 2016.
- [77] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data, SIGMOD '18*, pages 473–488, 2018.
- [78] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.
- [79] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 1629–1642, 2016.



# VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity

Qianxi Zhang<sup>1</sup> Shuotao Xu<sup>1</sup> Qi Chen<sup>1,\*</sup> Guoxin Sui<sup>1</sup> Jiadong Xie<sup>1,2</sup> Zhizhen Cai<sup>1,3</sup>  
Yaoqi Chen<sup>1,3</sup> Yinxuan He<sup>1,4</sup> Yuqing Yang<sup>1</sup> Fan Yang<sup>1</sup> Mao Yang<sup>1</sup> Lidong Zhou<sup>1</sup>  
<sup>1</sup>Microsoft Research Asia <sup>2</sup>East China Normal University

<sup>3</sup>University of Science and Technology of China <sup>4</sup>Renmin University of China

## Abstract

Approximate similarity queries on high-dimensional vector indices have become the cornerstone for many critical online services. An increasing need for more sophisticated vector queries requires integrating vector search systems with relational databases. However, high-dimensional vector indices do not exhibit monotonicity, a critical property of conventional indices. The lack of monotonicity forces existing vector systems to rely on monotonicity-preserving tentative indices, set up temporarily for a target vector's TopK nearest neighbors, to facilitate queries. This leads to suboptimal performance due to the difficulty to predict the optimal  $K$ .

This paper presents VBASE, a system that efficiently supports complex queries of both approximate similarity search and relational operators. VBASE identifies a common property, *relaxed monotonicity*, to unify two seemingly incompatible systems. This common property allows VBASE to circumvent the constraints of a TopK-only interface to achieve significantly higher efficiency, while provably preserving the semantics of TopK-based solutions. Evaluation results show VBASE offers up to three orders-of-magnitude higher performance than state-of-the-art vector systems on complex online vector queries. VBASE further enables analytical similarity queries that previous vector systems do not, and shows  $7,000\times$  speedup with 99.9% accuracy of exact queries.

## 1 Introduction

Recent advances in deep learning (embedding) models map almost all types of data (*e.g.*, images, videos, documents) into high-dimension *vectors* [60, 66, 88]. Queries on high-dimensional vectors enable complex semantic-analysis that was previously difficult if not impossible, thus they become the cornerstone for many important online services like search [25, 51], eCommerce [54], and recommendation systems [49, 53, 56, 84]. The “online” nature of these

services requires vector search to complete in milliseconds [31, 36, 42, 64]. Such a strict latency conflicts with the inherently high cost of exact search algorithm [28], which forces end-users to settle on approximate query results on high-dimensional vectors. With emerging new vector search applications, queries on vectors become increasingly more complex, which often involve hybrid search on both scalar and vector data (§2.1). This naturally motivates an integration of vector search systems and relational databases.

Vector search and database systems differ in their ways of using the index, a critical structure to speed up queries. An important property of conventional indices like B-tree [22] is *monotonicity*. This property ensures that a query can traverse the data-set guided by an index monotonically along a certain direction. This often avoids total data scan, therefore enables efficient query execution. However, it is prohibitively expensive for high-dimensional vector indices [5, 25, 43, 55, 57, 85] to preserve monotonicity, because of the curse of dimensionality [28]. Instead, they are often organized as a graph or cluster-based irregular structure, which follows monotonicity *approximately*. Traversing such vector indices does not guarantee a strict monotonic order in terms of distances to a target vector, but it enables a system to efficiently determine when it is *unlikely* for new traversals to reach closer vectors to a target than the current  $K$  ones. Therefore, modern vector indices only support approximate TopK, *i.e.*, to find  $K$  nearest neighbors approximately. A TopK query traverses a vector index for a sufficiently large number of steps, until it determines that a neighbor closer than the current  $K$  nearest ones is unlikely to be found.

To integrate vector search and database systems, existing vector database systems [76, 80, 86] choose to conform with *strict* monotonicity. To support similarity queries other than TopK, they first leverage TopK to collect  $K$  vectors, and sort the  $K$  vectors according to distances to a target vector, which sets up a *temporary index preserving monotonicity*. Complex relational operators can therefore execute on the temporary index in the traditional way. Consider the following vector search query, “find  $X$  number of products most similar to an

\*Corresponding author.



image but under a certain price”. A database planner would first run a vector search operator on the vector attribute of image embedding to find  $K$  nearest tuples, then apply a filter operator on the price attribute. But it is impossible to predict exactly how many tuples will pass the filter operator, which could be much less than  $K$ . Therefore this practice has the inherent difficulty of identifying the optimal  $K$  that produces exact  $X$  results. As a result, it resorts to either a setting of a conservatively large  $K$  or a trial-and-error of many  $K$ s, which both lead to suboptimal query performance.

In this paper, we present VBASE, a new system capable of efficiently serving complex online queries that involve both approximate similarity search and relational operators on scalar and vector data-sets. VBASE identifies *Relaxed Monotonicity* as the common property abstracted from the two seemingly different systems: vector search systems and relational databases. *Relaxed monotonicity* requires index traversals to only follow monotonicity *approximately*. We observe that state-of-the-art vector indices all follow relaxed monotonicity property in a two-phase pattern: an index traversal first locates the nearest region to a target vector approximately, and then moves away from the target region progressively in an approximate way. Based on the observation, we formally define *Relaxed Monotonicity* property, which abstracts the core index traversal pattern that most existing vector indices already have (§3.1). *Relaxed monotonicity* can be viewed as a generalized form of monotonicity, thus it is also applicable to conventional scalar indices, such as B-trees. Therefore, *Relaxed Monotonicity* can serve as the common foundation of vector search and database systems.

With relaxed monotonicity, VBASE builds a unified query execution engine to support a wide range of queries both on scalar and vector data, including queries across multiple heterogeneous indices. VBASE’s unified engine is based on a *Next* interface, instead of *TopK*, to support traversal in both vector and scalar indices. Meanwhile, the engine allows the derivation of a generalized termination condition from relaxed monotonicity to stop a query’s execution timely.

A unique characteristic of VBASE is that its relaxed-monotonicity-based query execution engine can *provably* achieve *equivalent* query results to those produced by *TopK*-only solutions of the *optimal  $\tilde{K}$*  (§3.3). This powerful property allows VBASE to circumvent the constraints of a *TopK*-only interface to achieve significantly higher efficiency, while preserving the semantics of *TopK*-based queries. In particular, based on the derived generalized termination condition, VBASE is able to detect the  $\tilde{K}$  during index traversal without an prediction of  $\tilde{K}$ . This allows VBASE to achieve similar performance to the well-optimized *TopK* vector search. For more complex queries than *TopK* searches, VBASE can achieve up to three order-of-magnitude lower average and tail query latency over state-of-the-art systems under similar result accuracy (i.e., same or even better recalls).

Moreover, with relaxed monotonicity, VBASE can even

Table 1: Online Similarity Query Support for Vectors

	S1	S2	S3	S4
ANN systems [25, 43, 46]	✓	✗	✗	✗
AnalyticDB-V [80]	✓	✓	✗*	✗*
PASE [86]	✓	✓	✗*	✗*
PostgreSQL [12]	✗*	✗*	✗*	✗*
Milvus [76]	✓	✓	✓	✗
Elasticsearch [4]	✓	✓	✗ <sup>o</sup>	✗

\*: Some systems can support these queries through exhaustive linear scan, but this cannot meet the requirements of online services.

<sup>o</sup>: Only support one inverted index and one vector index.

support approximate query types that previous systems do not, and show superior query performance and accuracy. For example, VBASE can finish a *join*-based vector query in 16 seconds with 99.9%+ recall rate, which is 7000× faster than a brute-force table scan.

In summary, we make the following contributions:

1. VBASE identifies and defines formally “*Relaxed Monotonicity*”, a property that reveals, for the first time, the core of well-designed vector indices and why they work effectively in practice.
2. VBASE builds a *Unified Database Engine* based on *relaxed monotonicity*, which enables powerful complex queries leveraging both vector and scalar data indices.
3. We prove that VBASE’s unified engine produces equivalent results to *TopK*-only methods using vector indices, with a much more efficient execution plan than that of *TopK*-only methods.
4. We implement VBASE based on PostgreSQL with 2000 lines of additional code, and show an end-to-end evaluation of eight complex SQL queries on a hybrid one million recipe data-sets [59] with both vector and scalar attributes.

We plan to make VBASE open-source to satisfy the emerging important vector analytic applications in the era of AI.

## 2 Background

### 2.1 Emerging Online Vector Queries

Vector has become a key form of data representation in the AI era. Deep learning has enabled a growing number of vector-centric online applications, including embedding-based retrieval [25, 87], face recognition [69], code retrieval [37], question answering [52, 63], Google Multisearch [7], Facebook near-exact duplicates detection [6], etc. More recently, AI applications have leveraged ChatGPT’s retrieval plugin [10] to convert their proprietary knowledge, personal documents, and chat contexts into vectors. This enables the retrieval of relevant vectors with price, category, location, or time constraints to construct prompts in the chat.

Traditional applications also benefit from vectors empowered by AI. For example, search engine turns web documents into both bag-of-words sparse vectors and deep learning em-

bedding dense vectors to improve the relevance of search results. And recommendation systems turn images, videos, and descriptions of items into different vectors. Combined with scalar data like item price and category, these vectors are used to enhance recommendation experiences.

All these call for a general system to run sophisticated vector and scalar queries efficiently. In summary, these vector scenarios can be categorized into the following types.

**S1: Single-vector TopK.** embedding-based retrieval [25], recommendation [87], and question answering [52, 63] essentially search a vector data-set for the  $K$  closest vectors, given a query vector. Such queries can be naturally expressed by a TopK operator on a single-vector column.

**S2: Single-vector TopKplus scalar attribute filtering.** There are also requirements to find TopK results under certain scalar attribute constraints. Google Multisearch [7] belongs to this category. It allows users to provide additional text hints during a similarity image search.

**S3: Multi-column TopK.** Some vector analytics require intersecting results of multiple TopK searches over different vector attributes. For example, image-recipe retrieval [68] is a recipe search on multi-modal data attributes of both (vectorized) ingredient keywords and a sample dish image. Recent work [70, 83] shows that multi-column TopK search can boost result quality in applications such as question-answering.

**S4: Vector similarity filter.** Similarity filtering is a typical vector analytics scenario. For example, face recognition [69] and Facebook’s near-exact duplicates detection [6] search for similar images (given an image) from a data-set with a similarity threshold. To support such applications, one could use vector filtering based on distance similarity between two images, i.e., distance-based range query.

All these vector query types have a strict latency requirement (e.g. milliseconds). Unfortunately, no existing systems can support all these online similarity queries comprehensively and efficiently (see Table 1).

## 2.2 The Division Between Databases and Vector Search Systems

Although databases can express the above queries through relational algebra, the division in the semantics between vector and conventional database indices makes it difficult to provide a unified system that efficiently runs various types of sophisticated online vector queries as shown in Table 1.

**Relational database.** A relational database is one of the most prominent tools to run sophisticated queries [16, 24, 29, 40]. In order to meet the low-latency “online” requirement, indices are widely adopted by databases to expedite query executions, such as B-tree [22], B+-tree [75] and more. These indices demonstrate *monotonicity*, a property that allows a query to traverse an index monotonically along a certain direction, e.g., in a descending or ascending order.

One of the most important types of online queries in the context of emerging vector scenarios is TopK query (§2.1). And a conventional database index can speed up TopK by traversing the index in the ascending or descending order and terminating the query as soon as it collects  $K$  results. This optimization applies to many TopK variants, such as TopK + filtering, and multiple-column TopK queries [33].

However, the effectiveness of such optimization relies on the assumption of monotonicity, which high-dimensional vector indices do not follow. We elaborate next.

**Approximate vector search.** The recent eruption of AI models has been generating a large and growing amount of high-dimensional vector data. For better learning representation, a vector can have hundreds of dimensions [60, 66, 88]. Due to the curse of dimensionality [28], no solutions can complete a high-dimensional vector query in sub-linear time. To address “online” scenarios, modern vector search systems resort to approximation to lower query latency dramatically (milliseconds) while maintaining a relatively high result accuracy (90%+ recall). These systems are often referred as *approximate nearest neighbor search* (ANNS) systems [5, 25, 43, 55, 57, 85].

Like relational databases, vector indices are adopted to facilitate approximate vector search. Representative vector indices are either organized as *partitions* (clustering-based [5, 17, 19, 25, 44, 45, 48, 90]), hash-based [30, 41, 79, 81, 85]), high-dimensional tree-based [23, 57, 62, 78]), or *neighborhood graphs* [32, 39, 43, 55, 58, 77]. The difficulty of locating a vector in the high-dimensional space forces these vector indices to optimize for approximate TopK. In a TopK query, index traversal is guided by a query vector  $q$  approximately towards the nearest neighbors tortuously based on the distance between  $q$  and some anchor points (e.g. cluster centroids, or sampled graph vertices). During the traversal, the direction to  $q$  may change dramatically, thus the process does not guarantee to approach  $q$  in every traversal step and the vector index traversal is not monotonic.

The lack of monotonicity in vector indices bars database systems from directly using vector indices to expedite queries, which is the primary source of *the division between databases and vector search systems*.

**TopK-based solutions to eliminate the division.** Because vector indices are optimized for TopK, ANNS systems expose only a TopK interface. To close the monotonicity gap between databases and vector search systems, the current practice is to use ANNS TopK interface to create tentative indices based on  $K$  vectors sorted according to the distance to the target vector. Such tentative indices preserve monotonicity, which enables fast vector query processing in databases [76, 80, 86].

However, TopK-based tentative index solutions are unsatisfactory. It is difficult, if not impossible, to predict the right size of  $\tilde{K}$  for the tentative index for queries, where a subsequent relational operator with a filter constraint can collect just the right number of results. This limitation universally applies

to TopK + filter queries, vector similarity filter queries, and more. Thus TopK-based tentative indices inevitably lead to choosing a conservatively very large  $K$  [80, 86] or performing trial-and-error with different sizes of  $K$  [76], both incurring excessive data accesses and computations.

### 3 VBASE Design

#### 3.1 Relaxed Monotonicity

Unlike conventional scalar indices, high-dimensional vector indices are designed for approximate TopK and do not follow monotonicity. Figure 1 shows the TopK traversal patterns of two popular vector indices, FAISS IVFFlat [5] and HNSW [89]. As illustrated, vector index traversal does not comply with monotonicity, the distance towards the target vector oscillates *unpredictably* as the traversal progresses. A lack of monotonicity in these vector indices bars relational databases from directly using them to expedite queries (§2.2).

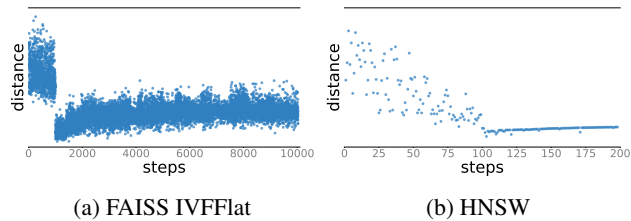


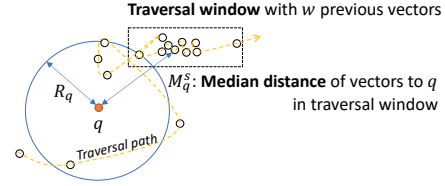
Figure 1: Traversal patterns of two vector indices.

**The Two-phase Vector Index Traversal Pattern.** Nevertheless, Figure 1 reveals a two-phase index traversal pattern for both vector indices. In the first phase, the index traversal approaches the target vector region approximately in spite of large oscillations in vector distances. In the second phase, the index traversal stabilizes and steadily departs from the target vector region in an approximate way.

This two-phase traversal pattern is common in most vector indices we examine. We believe that the essence of well-designed vector indices is an effective data-structure that embodies this traversal pattern implicitly. Thus a TopK search query could terminate early when it enters the second phase as further traversals are unlikely to identify more similar vectors.

**The Formal Definition of Relaxed Monotonicity.** Based on the two-phase traversal pattern, we can formally define *Relaxed Monotonicity* which identifies if a vector index traversal has entered the second phase. The definition is built upon the intuition of how a vector TopK search is executed internally.

Figure 2 shows such an intuition by illustrating the process of a general vector search for a query vector  $q$ . The dashed arrow in the figure shows an index traversal path with respect to  $q$ . Following the two-phase pattern, the query first approaches the *neighborhood* of  $q$  gradually. In a high-dimensional space, the neighborhood of  $q$  is defined by a neighbor sphere centered around  $q$ , illustrated as a circle in Figure 2. Afterward,



Neighbor sphere of a target vector  $q$  with a radius  $R_q$ , which contains  $E$  nearest vectors to  $q$ .

Figure 2: An Illustration of *Relaxed Monotonicity*'s intuition. A vector query  $q$  discovers  $q$ 's neighborhood with  $E$  nearest vectors along the progression of a traversal path.

the index traversal leaves the sphere and enters phase two, where the query can terminate in this phase.

Figure 2 suggests that, to determine whether it enters phase two, a query needs to understand  $R_q$ , the radius of the neighbor sphere centered around  $q$ , and whether  $M_q^s$ , the distance between the query's current index traversal position (denoted as traversal step  $s$ ) and  $q$ , is greater than  $R_q$ , i.e., it is traversing beyond the neighborhood of  $q$ .

Formally,  $R_q$ , the radius of  $q$ 's neighborhood, is defined as:

$$R_q = \text{Max}(\text{Top}E(\{\text{Distance}(q, v_j) | j \in [1, s-1]\})), \quad (1)$$

where  $\text{Top}E$  denotes the  $E$  nearest neighbors of  $q$  observed during the traversal, supposing that the traversal has reached step  $s$  so far. For a  $K$  nearest vector search query, it requires  $E \geq K$  in order to produce sufficient final results. In Figure 2, the  $E$  vectors within the circle are the nearest neighbors of  $q$  of all the  $s$  vectors visited so far. During an index traversal, the sphere's radius  $R_q$  would gradually decrease during phase 1, and becomes stable during phase 2.

Given the definition of  $R_q$ , the system needs to define  $M_q^s$ , the distance measurement between the target vector  $q$  and the current index traversal position, denoted as traversal step  $s$ .  $M_q^s$  is then used to determine whether the traversal enters phase 2, i.e., leaving the neighbor sphere.

Mathematically,  $M_q^s$  is defined as the median distance to  $q$  of all vectors traversed in the most recent  $w$  steps, i.e., the traversal window.

$$M_q^s = \text{Median}(\{\text{Distance}(q, v_i) | i \in [s-w+1, s]\}), \quad (2)$$

where  $\text{Distance}(q, v_i)$  denotes the distance between  $q$  and vector  $v_i$ , traversed in the past traversal window. Note that we use *median* instead of *mean* to disregard any outlier vectors in the traversal window, which has exceedingly large or small distances to  $q$  than others. For example, the two outlier vectors in the leftmost and rightmost positions in the traversal window shown in Figure 2.

Taking Eq.1 and Eq.2 together, we define *Relaxed Monotonicity* as:

**Definition 1 Relaxed Monotonicity**

$$\exists s, M_q^s \geq R_q, \forall t \geq s. \quad (3)$$



In other words, Def. 1 determines that a vector index follows *relaxed monotonicity* if there exists a certain index traversal step  $s$ , where all traversal steps  $t$  after step  $s$  transcends into a region ( $M_q^t$  as the region's distance to  $q$ ) that is outside  $q$ 's neighborhood sphere, defined by  $q$ 's  $E$  nearest neighbors.

**Importance of Relaxed Monotonicity.** Relaxed monotonicity is the key for the database to circumvent the inefficient constraint of TopK-only interfaces, and to generate an efficient execution with on-the-fly early termination. When subsequent database operators following the vector index scan can determine that vector index traversal has entered the second phase, one would know that we are veering away from the target vector steadily. In such cases, we could early terminate the query if sufficient results have been collected, because new tuples with closer vector attributes are unlikely to be found.

**Generality of Relaxed Monotonicity.** All mainstream vector indices listed in ANN Benchmarks [2] perform vector search using four general components: 1) *index traversal* to navigate the vector data-set; 2) *termination check* to detect query termination signal; 3) *monotonicity check* to determine if a query enters Phase 2; and 4) *priority queue* for keeping  $K$  nearest vectors so far. Often in ANNS indices, *monotonicity check* is a necessary condition for the *termination check*.

Although vector indices implement these four components in different ways, their monotonicity check satisfies Def. 1. For example, Figure 1 shows that index traversal patterns of IVFFlat and HNSW follow *relaxed monotonicity* obviously. And Def. 1's parameters, i.e., the traversal window  $w$  and the neighborhood of  $q$   $R_q$ , are able to capture the internal characteristics of index traversal patterns of these popular vector indices as well as conventional indices. Next, we elaborate on the setting of these parameters for representative indices.

- **Graph-based Vector Indices**, such as HNSW [89], follow the two-phase pattern using graph data-structures. In the first phase in Figure 1b, HNSW quickly navigates the traversal to the neighborhood of  $q$  through hierarchical coarse-grained to fine-grained navigating graphs. When it reaches the fine-grained graph, the traversal enters the second phase where it has found the neighborhood of  $q$  and departs away. Vector search using a graph-based index use best-first (BF) graph traversal from a fixed starting point. BF search maintains a *sorted candidate queue* with the size  $ef$ . This queue essentially represents the neighborhood sphere in Eq. 1 with  $ef$  vectors. Therefore  $E$  equals  $ef$  for HNSW. BF traversal explores the graph through the vectors in the candidate queue and expands the exploration by visiting their neighbors. If a neighbor is *unvisited* and its distance to the target vector  $q$  is smaller than the farthest vector in the sorted queue (i.e.  $R_q$ ), the traversal replaces the farthest vector with the new one and resorts the queue. Because the traversal only compares the traversed vector itself with  $R_q$ , the traversal window  $w$  in Eq. 2 equals one.
- **Partition-based Vector Indices**, such as FAISS IVFFlat [5] and SPANN [25], divide vectors into multiple clus-

ters where nearby vectors are conglomerated. During the traversal in the first phase in Figure 1a, IVFFlat traverses over the centroids to identify the  $m$  closest clusters, and then in the second phase it goes over the vectors in  $m$  clusters and terminates when all vectors in  $m$  clusters are traversed, which indicates Eq. 3 of *Relaxed Monotonicity* has been satisfied after the vector search visits  $m$  clusters. With this observation,  $E$  in Eq. 1 is set to  $K$  of the TopK query, and the traversal window  $w$  in Eq. 2 is set to the number of total vectors in  $m$  clusters.

- **Scalar Indices**, such as B-Tree, follow strict monotonicity. It is a special case of relaxed monotonicity, where  $w$  and  $E$  are both set to 1 and Eq. 3 is always true.

Note that it is our observation that well-designed indices should satisfy *Relaxed Monotonicity*. VBASE abstracts and formalizes this property that most indices already preserve, and encapsulates it with mathematical terms such as  $E$  and  $w$  in Eq. 1 and 2. It is the responsibility of individual indices to guarantee relaxed monotonicity by tuning the corresponding hyperparameters like  $ef$  and  $m$ , which can be transformed to  $E$  and  $w$ , as discussed above.

## 3.2 Unified Query Execution Engine

With relaxed monotonicity, VBASE builds a unified query execution engine modeled after a traditional database engine with minimum changes. VBASE's unified engine is built on Volcano Model (i.e. Iterator Model) [35], where 1) a relational operator in a given query produces a stream of tuples iteratively that are consumed by downstream operators, and 2) the iterative execution stops if a termination condition is met.

**Iterative Execution Model.** VBASE fully complies with the Volcano Model. It reuses the traditional `Open`, `Next`, and `Close` interfaces to leverage index traversal so that no change is required for conventional indices. For vector indices that traditionally expose TopK interfaces only, VBASE performs a simple adaptation to expose their internal index traversal process, to conform to VBASE's `Next` interface. We will discuss the details of implementing `Next` for vector indices in §4.2.

**Generalized Termination Condition Check.** VBASE modifies the termination condition based on relaxed monotonicity. In particular, VBASE extends the original termination condition with *relaxed monotonicity check*. In addition to the original query termination condition, VBASE performs relaxed monotonicity check by inspecting Eq. 3. Because VBASE requires a vector index guarantees *Relaxed Monotonicity*, an inspection of Eq. 3 could be reduced to  $M_q^s > R_q$ , where *relaxed monotonicity* checks beyond step  $s$  are all assumed to be true.

The query execution would only stop if both the original termination condition and *relaxed monotonicity* check were passed. Please note for the traditional index, the *relaxed monotonicity* check is always true, which reduces to the termination check of the convention iterative model.



Next, we describe how VBASE’s termination conditions work for TopK and distance-based ranger filter, two operators used by vector queries.

- **OrderBy with limit:** In traditional databases, TopK is usually expressed by an OrderBy operator with limit  $K$ . The traditional TopK query terminates immediately once  $K$  results have been collected, because all indexed tuples are ordered. For an approximate TopK query on vectors, VBASE need to check if the relaxed monotonicity check (i.e., reduced Eq.3) is passed, in addition to collecting  $K$  vectors. When *relaxed monotonicity* check is true, the index traversal has entered phase 2 (§3.1), which indicates it is veering away from the target vector steadily. And we can terminate the query after collecting  $K$  nearest vectors, because it is unlikely to find new vectors closer than the collected ones.
- **Range filter:** Distance-based range filter returns tuples whose values or distances to a target are within a range  $R$ . For a conventional query, the query stops when a tuple being traversed goes beyond range  $R$ , because monotonicity guarantees we have visited all tuples within  $R$ . For a vector query, the execution only stops if both a vector along the traversal path exceeds distance  $R$  and the relaxed monotonicity check is passed, which indicates we are in a stable phase (phase two) moving away from a target vector.

**Advantages of a Unified Engine.** The unified engine enables VBASE to preserve full compatibility with traditional databases and supports all the vector query types discussed in §2.1. It also creates new optimization opportunities for vector queries. For example, instead of filtering after TopK, VBASE can perform filtering during index traversal with flexible termination conditions (for TopK or range query). This optimization is one of the key reasons VBASE outperforms TopK-based solutions (§5.3). Moreover, the unified engine allows the incorporation of a refined NRA algorithm [33], which can significantly improve the performance of multi-column vector query (§4.4).

Interestingly, VBASE’s unified engine also supports vector Join. Although not an online query, vector Join is useful in scenarios such as document auto-tagging [26, 72], where a small labeled (tagged) document set is used to identify a tag for each document in a large unlabeled document set by finding the document pair with the closest document embeddings (vectors). This can be thought of as running a Join operator on a document table and a label table with a distance-based match, which can be achieved in VBASE by an index join based on a range filter. Because such a Join can only be achieved by a full table scan in existing solutions, VBASE can outperform the baseline by over  $7000\times$  (§5.3).

### 3.3 Result Equivalence

In this section we demonstrate a powerful property of VBASE’s unified query execution engine based on *Relaxed Monotonicity*, that it produces the equivalent results as a TopK

method based on a tentative monotonicity-preserving index with the optimal  $\tilde{K}$ . The optimal  $\tilde{K}$  is the minimal  $K'$  for the index traversal to satisfy  $K$  results in a TopK query. As  $K'$  is the minimal satisfactory value, the query latency is minimized. We rely on the quality of individual indices to ensure recalls.

Next, we formally prove the result equivalence for TopK+filter and range filter [20] queries, which are major types of similarity searches supported by existing TopK-based vector systems (Table 1). The proof also reveals the reason of VBASE’s superior performance.

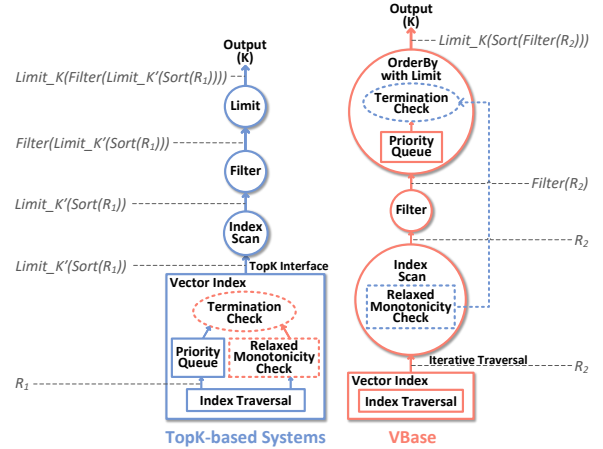


Figure 3: Result Equivalence

**TopK +filter.** To find  $K$  vectors matching the filter, a TopK-based system first collects  $K'$  vectors by calling TopK ( $K'$ ) and sorts the collected  $K'$  vectors. To achieve this, the system needs to traverse  $R_1$  vectors through the underlying vector index. The query then runs on the sorted  $K'$  vectors by applying the filter and the  $K$  limit operators, producing the final results, denoted as  $r_1$ . Eq.4 formulates the above process, illustrated on the left in Figure 3.

$$r_1 = \text{Limit}_K(\text{Filter}(\text{Limit}_{K'}(\text{Sort}(R_1)))). \quad (4)$$

We define *filter\_selectivity* as the ratio of the output set on the input set of the filter operator. Eq. 4 can then be transformed to:

$$r_1 = \text{Limit}_{K''}(\text{Filter}(\text{Sort}(R_1))), \quad (5)$$

where  $K'' = \min(K, K' \times \text{filter\_selectivity})$ .

Assuming the TopK-based system can predict the optimal  $\tilde{K}$ , i.e.,  $K' = \tilde{K} = K / \text{filter\_selectivity}$ , execution will get exactly  $K$  results, and Eq. 5 reduces to Eq. 6

$$r_1 = \text{Limit}_K(\text{Filter}(\text{Sort}(R_1))). \quad (6)$$

In comparison, VBASE traverses  $R_2$  vectors via the same vector index as the TopK-based system, and gets results  $r_2$ . Eq. 7 formulates this process, shown on the right of Figure 3.

$$r_2 = \text{Limit}_K(\text{Sort}(\text{Filter}(R_2))). \quad (7)$$

We show in §3.2 and §4 that TopK-based systems and VBASE use the same vector index, follow the same index traversal algorithm, and are based on the same relaxed monotonicity check to terminate queries. Therefore both systems traverse the exact same set of tuples, i.e.,  $R_1 = R_2$ .<sup>1</sup> Because  $R_1 = R_2$  and *Filter* and *Sort* are commutative, from Eq. 6 and 7 we conclude  $r_1 = r_2$ . **Q.E.D.**

It is difficult for a TopK-based solution to predict  $K'$  to get both correct results and high query efficiency. If  $K' \times filter\_selectivity < K$  in Eq.5, the system cannot get enough results, leading to poor accuracy. If  $K' \times filter\_selectivity > K$  in Eq. 5, the result is accurate at the expense of splurging extra index traversal. Some TopK-based system [76] performs trial-and-error with many values of  $K'$  until  $K' \times filter\_selectivity \geq K$ , which results in excessive duplicated data access and processing. In contrast, VBASE determines  $\tilde{K} \times filter\_selectivity = K$  on-the-fly, therefore achieving both high query accuracy and performance.

**Range Filter Query.** A range filter query based on TopK can be formulated as

$$r_1 = Filter(Limit\_K'(Sort(R_1))), \quad (8)$$

Where  $R_1$  stands for the traversed vectors by the underlying TopK primitive. Similar to Eq.4 vs. Eq.5, Eq. 8 can be transformed to

$$r_1 = Limit\_K''(Filter((Sort(R_1)))), \quad (9)$$

where  $K'' = K' \times filter\_selectivity$ .

Under the assumption of optimal  $\tilde{K}$ , assuming the query produces  $T$  vectors, we have  $\tilde{K} = K' = T / filter\_selectivity$ . Based on  $\tilde{K}$ , Therefore,

$$r_1 = Limit\_T(Filter(Sort(R_1))) = Filter(Sort(R_1)). \quad (10)$$

In comparison, VBASE traverses  $R_2$  vectors via the same index and gets results  $r_2$ , which can be formulated as

$$r_2 = Filter(R_2). \quad (11)$$

Since the traversal algorithm and the termination condition are exactly the same as the TopK-based solution, both systems visit the same set of vectors, i.e.,  $R_1 = R_2$ . As the filter conditions are not sensitive to order,  $r_1 = r_2$ . **Q.E.D.**

## 4 VBASE Implementation

### 4.1 Relaxed Monotonicity Check

We implement a common relaxed monotonicity check for all vector indices based on Definition 1 in §3.1. Specifically, we implement two queues to track the current traversal state: 1)

<sup>1</sup>We assume vector index traversal is deterministic, true for most vector search systems.

a priority queue with size  $E$  called `smallestQueue`, to keep the visited nearest neighbors of a target vector  $q$  during the traversal; 2) `recentQueue` of size  $w$  to track the most recent traversal window. When a new vector  $v$  is visited via index traversal, `smallestQueue` and `recentQueue` are updated accordingly. And the relaxed monotonicity check is performed by calculating the current traversal state according to Eq.(3) based on vectors in `smallestQueue` and `recentQueue`.

Note that  $E$  and  $w$  are sensitive to data distribution and specific indexing algorithms. Increasing them tends to improve query accuracy at the expense of longer latency. In practice, they can be tuned to trade off query accuracy and latency.

### 4.2 Query Execution Engine

VBASE's unified query engine is implemented based on PostgreSQL, with minor extensions to modules regarding index traversal and termination conditions.

**Vector index integration.** Existing high-dimensional vector indices only expose TopK interface and keep the index traversal and relaxed monotonicity check internally to the system. VBASE re-architects the vector indices systems by exposing the internal index traversal algorithms with `Open`, `Next`, and `Close` interfaces, which can then be integrated into Volcano Model seamlessly.

VBASE has incorporated several state-of-the-art vector indices, including HNSW [89], IVFFlat [5] and SPANN [25], where SPANN is shown effective for billion-scale vector datasets. Next, we introduce the integration of HNSW and IVFFlat, a graph-based index and a partition-based index, respectively. Other algorithms can be integrated in a similar way.

HNSW [89] is a graph-based vector index consisting of hierarchical neighborhood graphs where the upper-layer graph keeps coarse-grained samples of the lower-layer graph. A query traverses the graphs from upper-layer to lower-layer following the best-first manner. The approximate nearest point found in the upper-layer graph is the entry point of the lower-layer graph. In VBASE, we remove the implementation regarding TopK, e.g., a priority queue to record the top  $k$  results, and only keep states necessary to carry on the index traversal algorithm. The relevant states include a bitmap to record previously visited vectors, the current vector being visited, and the candidate vectors to be visited next. These states will be kept during the query life cycle. To initiate a query on a vector index, VBASE calls `Open` to search on high-layer graphs. During query execution, each call to `Next` will return the current closest unvisited node, records it, and expands its neighbors into candidate vectors in the state. The state will be cleared in `Close` function. Overall, we modify less than 200 lines of code to integrate HNSW.

IVFFlat [5] is a partition-based index, which clusters vectors into lists and chooses the centroid as the representative of each list. In the `Open` interface used to initiate index traversal, VBASE sorts all the lists from near to far based on the dis-

tance between the target vector and the centroids. Upon calls to `Next`, the vectors in the corresponding nearest lists are read one by one. The query execution state in a partition-based index includes sorted lists and the current read position, which will be destroyed by a `Close`.

**Index scan operator.** We add a new “vector index” type using the index extension interface in PostgreSQL [12] to implement index scan. It forwards function calls to `Next` to the underlying vector index within the iterative interface. We use `array` to store high-dimensional vectors in the table and record their tuple addresses in the table as the vectors’ metadata in the index. Once a vector is read from the underlying vector index, its metadata will also be returned so that VBASE can find the corresponding tuple in the main table.

Note that the relaxed monotonicity check described in §4.1 is implemented in the index scan operator so that multiple indices do not need to duplicate the implementation.

**OrderBy with limit.** VBASE implement `TopK` using `OrderBy` with `limit` plus an index scan operator. The system uses a priority queue to keep the candidate results. The `TopK` query terminates once the index traversal passes the relaxed monotonicity check in the upstream index scan operator and  $K$  vectors have been filled in the priority queue.

Note that vector indices are used for similarity queries, i.e. search closest vectors to the target vector. If a user would like to query the farthest `TopK` results from the target vector, the distance calculation method needs to be reversed before creating the indices.

**Range filter and Join.** VBASE implements an efficient range filter by concatenating it with an index scan operator. Only vectors passing the distance filter condition can be returned to the subsequent operators. The index traversal stops when the distance between the current vector to the target vector is larger than the filter constraint and the relaxed monotonicity check is passed in the index scan operator.

With the support of distance filter, VBASE can even support `Join` on high-dimensional vectors, which previous vector systems cannot support efficiently. Semantic-based join has been widely used in document auto-tagging [26, 72], which assigns one or multiple labels for each unseen document by finding the closest label embeddings to a document embedding. Previous systems can only support `Join` by brute-force table scan. VBASE executes a `Join` by nested-loop with index search, which outperforms existing systems by 7000× faster with 0.999 recall accuracy in our experiment.

**More complex queries.** The combination of the above operators can be used to support more complex queries efficiently.

### 4.3 Query Planning

Complex queries often require effective cost estimation on various query plans. In general, it includes vector algebra computation (e.g., distance calculation), selectivity estimation

in case the query contains filters, and index scan cost.

**Vector computation.** Traditional databases estimate the cost of scalar data computation using a constant value  $t$ , e.g.,  $t = 0.0025$ . But vector computation is more expensive, it involves the calculation of the distance between vectors, which is proportional to the number of dimensions. Thus VBASE models the cost of vector computation  $t_v$  as:

$$t_v(dim) = t \cdot c \cdot dim,$$

where  $t$  is a predefined value representing the cost of scalar operation,  $c$  is the coefficient related to SIMD optimizations for vector computation, and  $dim$  denotes vector dimension.

**Selectivity estimation.** If a query contains a filter, the query should estimate selectivity, the ratio of tuples that will pass the filter. VBASE relies on sampling-based methods to measure the distribution of high-dimensional vectors [67, 82]. Specifically, VBASE uniformly samples vector data at a ratio and stores the sampled vectors in the metadata of the database. Given a query  $q$ , it applies the filter on the sampled data to estimate the selectivity  $Sel$  on the full vector data-set.

$$Sel_{sample}(q) \approx Sel_{full\_data}(q).$$

In our experiments, setting the sample rate to 0.001 can produce a good estimation with q-error < 1.1 in most cases while incurring only a tiny extra latency (<1ms). More details will be presented in §5.5.

**Index scan cost estimation.** This includes start-up cost and traversal cost. The start-up cost is the cost to locate the region nearby the target vector before returning vector data; Traversal cost represents the cost to iterate over the matched tuples through the index. For each index traversal step, the cost  $C_{step}$  includes  $t_{IO}$ , the IO cost to fetch the index data from disk, and  $t_v(dim)$ , the cost to calculate the distance:  $C_{step} = t_v(dim) + t_{IO}$ . For partition-based indices like IVFFlat and SPANN, the index scan cost  $C_p$  is:

$$C_p = N_c \times C_{step} + \max(\lceil Sel(q)N/N_p \rceil, m) \times N_p \times C_{step},$$

where  $N$  is the table size,  $N_c$  is the number of centroids,  $N_p$  is the average number of data per partition, and  $m$  is the number of partitions the index traversal algorithm requires to traverse for *relaxed monotonicity* check.

The scan cost,  $C_g$ , of graph-based indices like HNSW is:

$$C_g = N_{start} \times C_{step} + \max(Sel(q)N, N_E) \times R_{iter} \times C_{step},$$

where  $N_{start}$  is the number of steps to traverse upper-layer graphs in `Open` function of HNSW,  $N_E$  is the number of steps to satisfy *relaxed monotonicity* check, and  $R_{iter}$  is the average times of distance function called per step to reach next point.  $N_{start}$ ,  $N_E$  and  $R_{iter}$  are dependent on the hyper-parameters of the index and the distribution of data, which can be automatically estimated by sampling, and this process can be embedded into databases’ `Analyze` routine.

## 4.4 Multi-Column Scan Optimization

To support multi-column vector queries, TopK-based systems can only perform multi-column scan based on the multiple sets of sorted vectors collected by TopK. For example, Milvus performs NRA algorithm [33] for multi-column scan based on TopK. It doubles  $K$  and re-executes the query if the previous results are insufficient. Every attempt to a larger  $K$  is an independent traversal over the underlying vector index. This introduces excessive vector access and computation.

In contrast, VBASE implements NRA algorithm [33] natively based on the index scan operator, thus avoiding the repetitive execution of NRA. The NRA algorithm traverses each vector index in a round-robin manner. We observe that round-robin might not be an efficient choice. In the vector search scenario, different vector indices can return results of different quality, especially when the ranking function is summation with unequal weights from different indices. Figure 4 shows the results of such a case, where the round-robin method will unnecessarily traverse excessive low-quality vectors (i.e., dots in Figure 4).

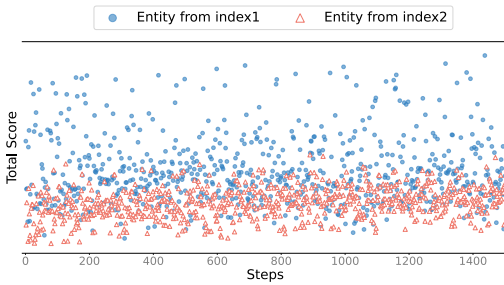


Figure 4: Index traversal pattern for a 2-index vector query on Recipe1M. The total score is a summation of two vectors’ distance with a weight ratio of 1:2. Lower score means closer to the target vector. Blue dots and red triangles represent the total scores of entities using index1 and index2.

This observation leads us to a new index scan algorithm that scans through high-quality indices more frequently, i.e., triangles in Figure 4, so that the query can terminate earlier with even more accurate results. Such a non-uniform traversal manner may trap in local optima. In Figure 4, a greedy algorithm may prefer to visit index2 only. But the figure shows that index1 does have good quality vectors occasionally.

To balance exploration and exploitation, we use both local and global information to guide the index traversal (See Figure 5). Our approach divides the traversal process into several rounds and adds a traversal decision module. It maintains a local priority queue to store the last round’s results. This local information helps us identify which index is more likely to return better results so we can visit it more in the next round. To avoid being trapped in local optima, the decision module also stores the average score of all traversed entities for each index (i.e.,  $avg_i$  for  $index_i$ ) and updates them each round. Based on this global information, We additionally tra-

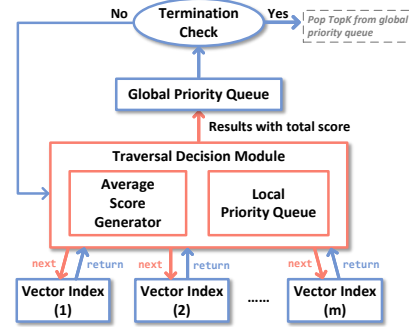


Figure 5: Overview of multi-column traversal optimization, assuming there are  $m$  indices.

verse each index  $W_i = \left\lceil n_2 \times \frac{1/avg_i}{\sum_{j=1}^m 1/avg_j} \right\rceil$  times in this round

where  $n_2$  is a hyper-parameter. Therefore, the high-quality index (with low  $avg_i$ ) will be traversed more times while we can still ensure traversing the low-quality index (with high  $avg_i$ ) at least once in each round. Table 7 in §5.3 highlights the benefit of this approach.

## 5 VBASE Evaluation

In this section, we evaluate VBASE in comparison with other state-of-the-art vector search systems and vector-enabled databases based on TopK, and demonstrate VBASE has superior performance and accuracy on vector similarity queries.

### 5.1 Evaluation Benchmark

A lack of a comprehensive relational benchmark for complex vector applications necessitates us to create a vector benchmark to compare VBASE with various vector-similarity-enabled systems. The use of approximate vector processing also needs the benchmark to define new evaluation metrics.

**Vector-scalar relational data-set.** Because current vector search [2,3] and database benchmarks [13,14] have either vector or scalar data-sets *but not both*, we extend Recipe1M [68] to generate vector and scalar hybrid data-sets. Recipe1M data-set is a collection of more than 1 million recipes, each containing ingredients, cooking instructions, and a set of images of the finished dish.

Our evaluation data-set is organized as two tables: *Recipe Table* and *Tag Table*. Their schemas are shown in Table 2 and 3, respectively.

Table 2: Schema of Recipe Table

Column Name	Data Type	Example
recipe_id	identifier	1
images	list of strings	["data/images/1/0.jpg", ...]
description	text	[ingredients] + [instruction]
images_embedding	vector	[0.0421, 0.0296, ..., 0.0273]
description_embedding	vector	[0.0056, 0.0487, ..., 0.0034]
popularity	integer	300



Table 3: Schema of Tag Table

Column Name	Data Type	Example
id	identifier	1
tag_name	text	“salad”
tag_vector	vector	[0.0137,0.0421,...,0.0183]

*Recipe Table* stores 330,922 recipes from Recipe1M<sup>2</sup>. As shown in Table 2, *Recipe Table* inherits *recipe\_id* and *recipe\_images* URIs from Recipe1M as two attributes. We merge Recipe1M’s ingredients and instructions as a single string attribute called *description*.

In addition to the original data from *Recipe1M*, *Recipe Table* has two *vector* attributes, *images\_embedding* and *description\_embedding*. They are two 1,024-dimensional vector embeddings of recipe images and descriptions based on the cross-modal embedding model from [68]. We also extend the recipe item with an additional scalar attribute: *popularity*, a random integer in the range [0,10000].

*Tag Table* samples 10,000 recipes from Recipe1M, and assigns of *tags* for them. As shown in Table 3, *Tag Table* has scalar attributes of *id* and *tag\_name*. Attribute *tag\_name* is a set of strings of manually assigned tags (e.g. dessert, main course, salad, pizza, etc), and *id* is a unique integer assigned to the tag set. Each *Tag Table* row also has a *tag\_vector*, which is a 1,024-dimensional *images\_embedding* of a recipe using the same embedding model of the *Recipe Table*.

**Vector similarity queries in SQL.** We designed 7 SQL queries to emulate various vector online application scenarios (§2.1). In particular, we also designed Q8 which runs an analytic join query based on vector similarity match. These 8 relational queries cover most SQL operators of Projection, Index Scan, Sort with Limit, Filter, Join, which are important for online queries over vector and scalar data-set.

- Q1: Single-Vector TopK.

```
SELECT recipe_id FROM Recipe
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) LIMIT 50;
```

- Q2: Single-Vector TopK + Numeric Filter.

```
SELECT recipe_id FROM Recipe
WHERE popularity <= ${p_popularity}
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) LIMIT 50;
```

- Q3: Single-Vector TopK + String Filter.

```
SELECT recipe_id FROM Recipe
WHERE description NOT LIKE "%${p_ingredient}%"
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) LIMIT 50;
```

- Q4: Multi-Column TopK.

```
SELECT recipe_id FROM Recipe
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
    description_embedding, ${p_description_embedding})
LIMIT 50;
```

<sup>2</sup>We remove those in the 1 Million recipes that miss any of the ingredients, cooking instructions, and related images.

- Q5: Multi-Column TopK + Numeric Filter.

```
SELECT recipe_id FROM Recipe
WHERE popularity <= ${p_popularity}
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
    description_embedding, ${p_description_embedding})
LIMIT 50;
```

- Q6: Multi-Column TopK + String Filter.

```
SELECT recipe_id FROM Recipe
WHERE description NOT LIKE "%${p_ingredient}%"
ORDER BY INNER_PRODUCT(images_embedding,
    ${p_images_embedding}) + WEIGHT * INNER_PRODUCT(
    description_embedding, ${p_description_embedding})
LIMIT 50;
```

- Q7: Vector Range Filter.

```
SELECT recipe_id FROM Recipe
WHERE INNER_PRODUCT(images_embedding, ${p_images_embedding})
    <= ${D};
```

- Q8: Join.

```
SELECT Recipe.recipe_id, Tag.tag_name
FROM Recipe JOIN Tag
ON INNER_PRODUCT(Recipe.images_embedding, Tag.tag_vector)
    <= ${D};
```

We ran these 8 queries on VBASE and compared them with query processing in other systems. For each query, we generated 10,000 substitution parameters to cover various query conditions. In particular, we set *K* to 50 for TopK queries as in the experiment of Milvus [76]. We also designed the numeric filtering constraints to cover both high and low filtering selectivities. *p\_popularity* is incremented from 1 to 10000. *p\_ingredient* is sampled from ingredients keywords in Recipe1M. *WEIGHT* is 1. *D* is 0.1 in Q7 and 0.01 in Q8.

**Evaluation metrics.** Vector query executions are approximate, hence we evaluate both query accuracy and performance in terms of *recall* and *latency*, respectively. Recall is a new metric to conventional database evaluations. It evaluates query accuracy against the ground truth. Recall has been widely used in approximate vector search systems [2, 3, 76, 86]. They only evaluate recall because for TopK queries, recall and precision are the same as long as a system returns *K* results. For other queries with range filter constraints, precision will always be 1 if the system obeys the constraints. Therefore, we use recall to represent query accuracy. For each query, we calculate the average recall of the query results of all substitution parameters. For each query in Q1-Q7, we measure the *average, median, 99th percentile* latency from the execution results of all substitution parameters. For Q8, we execute it 3 times and measure the *average* execution time.

## 5.2 Experiment Setup

**Evaluation platform.** All evaluations run on an Azure VM, *Standard\_F64s\_v2* [1], with 64 v-CPU and 128 GiB memory running Linux Ubuntu 20.04 LTS. All queries run individually to avoid interference from other queries.

**Baseline systems.** We compare VBASE with the state-of-the-art vector search and database systems that support vector similarity queries.

**Vector search baselines:** We choose *Milvus* [76] and *Elasticsearch* [4] as our vector search baselines. Since they do not support SQL interface, we hand code our benchmark queries. We also implement Iterative Merging algorithm as claimed in Milvus paper [76] to enable multi-column TopK queries, although unavailable in its open-source code [8]. For Elasticsearch we use the version implemented by Open Distro (version 1.13) [9], which supports HNSW index.

**Database baselines:** For databases, we use the open-source PASE [11, 86] that implements the TopK-only solutions using tentative indices, and extended its maximum dimension support from 512 to 1024. We also run queries on *PostgreSQL* (version 13) [12] as a baseline to show the performance of traditional databases in performing vector similarity queries.

**Common index settings.** VBASE and baseline systems all use HNSW [89] with the same vector index settings ( $M = 16, ef\_construction = 200, ef\_search = 64$ ). HNSW is the only vector index supported by PASE, Milvus, and Elasticsearch in common. Since HNSW index is kept in memory when it is used, in order to better compare the performance results of the index-based execution process without being affected by the caching strategy of the main table adopted by different systems, we also save the main table data in memory. All the database baselines and VBASE also created a B-tree index on *popularity* column to expedite numeric filtering.

### 5.3 Evaluation Results

**Overview.** Table 4 summarizes the overall evaluation results. We can see that each baseline system based on approximate vector indices (except PostgreSQL) can only process some of the 8 queries, while VBASE can process all of them.

Although PostgreSQL can process all queries and produce exact results, it uses a brutal force scan with a much higher query latency than the rest of the systems. The  $1000\times$  lower performance of PostgreSQL than other approximate systems makes it irrelevant to address the low-latency “online” scenarios. We run queries on PostgreSQL mostly to get ground truth to calculate the query result accuracy of other systems.

VBASE’s query performance on TopK (Q1) is similar to or better than baseline systems because they essentially run the same algorithm in different implementations. For queries that are more complex than Q1, VBASE outperforms all baseline systems by  $100\times - 1000\times$  because VBASE can determine the optimal  $\tilde{K}$  on-the-fly and others have to try different  $K$ s to get sufficient results. While VBASE produces superior query performance, it can also achieve high recall similar to or even higher than approximate queries on baseline systems.

Next, we discuss each query’s evaluation result in detail.

**Q1 – Single vector TopK.** Q1 is a TopK query without any

filters. All approximate systems including VBASE in our evaluation run the same algorithm and produce identical results, therefore having the exact same recalls.

Nevertheless, we can see variations in Q1 latency from different systems. The reasons for performance variations are two-fold. The first reason is that these systems are implemented in different languages (Milvus/Elasticsearch vs. PASE/VBASE). For example, Milvus are implemented in GoLang and C++, and Elasticsearch is implemented in Java and C++. In comparison, PASE and VBASE are written in C. In general, we can see implementation in C outperforms other high-level language implementations by  $2 - 10\times$ .

The second reason for performance variation (PASE vs. VBASE) is that VBASE needs to fetch slightly more tuples than PASE. Although following the same algorithms and traversing the same amount of vectors in the index, VBASE follows an Iterator Model, which fetches every corresponding tuple from the main table during index traversal. PASE’s implementation visits vectors in the index and only fetches the  $K$  tuples after getting TopK vectors in the index. As a result, VBASE performs slightly worse, 2.8% slower than PASE in terms of average and 99 percentile latency.

**Q2-3 – Single vector TopK with scalar filter.** Q2 and Q3 are vector similarity queries with filtering on scalar attributes. Q2 and Q3 differ in their filter predicates, where Q2 uses numeric filtering on the integer attribute *popularity*, and Q3 runs string filtering based on a regular expression. All baselines support Q2 and Q3, with an exception of Milvus for Q3 because it does not support string data type.

All approximate baselines run a *single shot* of  $K'$ . Based on different guesses of  $K'$ s, baseline systems produce different results. Elasticsearch undershot  $K$  for Q2 and Q3, therefore cannot produce sufficient results and has low query accuracy. Even though Elasticsearch has fewer data traversals than the rest of the approximate systems with higher accuracy, its query latency is still the worst, possibly due to system inefficiency like in Q1. PASE and Milvus overshoot  $K$  and produce high result accuracy like VBASE, but they have longer query latency because they traverse more data than VBASE.

Q3 has a fixed filter selectivity of around 0.9. For Q2 we have 10,000 queries of different parameters with uniform distribution of filter selectivity from 0 to 1. We compared the evaluation results of two best approximate systems VBASE and PASE for Q2, under three representative *filter\_selectivity* values (0.03, 0.3, 0.9) from low to high (Table 5). We found that different filter selectivities result in different optimal  $\tilde{K}$ s: the lower the selectivity, the more data a system needs to examine, therefore larger  $\tilde{K}$ . Because  $\tilde{K}$  is dynamic, PASE’s static guess of  $K'$  cannot produce constantly high recalls under all filter selectivities. A conservatively large guess of  $K' = 10,000$  can produce near-exact results by PASE, however, its query performance deteriorates dramatically. We also present the average  $\tilde{K}$  and standard deviation under different filter selectivities in Table 5.  $\tilde{K}$  varies for different filter se-

Table 4: 8 Queries Result Overview (Latency: ms)

System	Q1:Single-Vector TopK				Q2:Single-Vector TopK+Numeric Filter				Q3:Single-Vector TopK+String Filter				Q4:Multi-Column TopK			
	Recall	Latency			Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th		average	median	99th
PostgreSQL	1	2,980.1	3,021.7	3,133.6	1	1,108.3	1,124.1	2,286.2	1	4,322.2	3529.3	9,953.0	1	5,610.0	5,604.7	5,769.8
PASE	0.9949	<b>4.8</b>	<b>3.5</b>	<b>5.1</b>	0.9987	29.3	28.7	61.7	0.9982	13.2	10.7	17.9	-	-	-	-
Milvus	0.9949	9.4	9	12.7	0.9919	33.7	23.9	121.4	-	-	-	-	0.9041	6,696.4	8,349.3	9,299.0
Elasticsearch	0.9949	43.1	41.8	48.9	0.5010	97.9	98.1	118.1	0.8378	79.9	90.0	100.9	-	-	-	-
VBase	0.9949	4.9	3.9	5.3	<b>0.9989</b>	<b>11.7</b>	<b>6.3</b>	<b>51.7</b>	<b>0.9983</b>	<b>7.9</b>	<b>6.7</b>	<b>10.4</b>	<b>0.9696</b>	<b>19.8</b>	<b>18.4</b>	<b>46.4</b>
System	Q5:Multi-Column TopK+Numeric Filter				Q6:Multi-Column TopK+String Filter				Q7:Vector Range Filter				Q8:Join			
	Recall	Latency			Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th		average	median	99th
PostgreSQL	1	1,192.9	1,234.4	2,343.6	1	6,543.2	5,996.3	16,734.6	1	8,244.9	8,212.6	8,641.6	1	129,051,273.9	-	-
PASE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Milvus	0.9691	12,637.9	5,617.4	36,887.9	-	-	-	-	-	-	-	-	-	-	-	-
Elasticsearch	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
VBase	<b>0.9805</b>	<b>35.8</b>	<b>24.9</b>	<b>160.7</b>	<b>0.9626</b>	<b>21.6</b>	<b>18.3</b>	<b>64.8</b>	<b>0.9840</b>	<b>10.8</b>	<b>2.2</b>	<b>168.9</b>	<b>0.9992</b>	<b>16,335.9</b>	<sup>1</sup>	<sup>1</sup>

<sup>1</sup> We have only run one query parameter for Q8, so average, median and 99th percentile latency are the same.

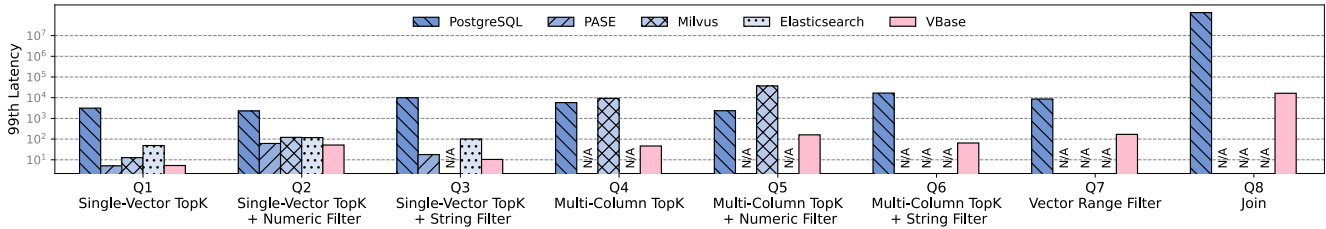


Figure 6: 99th Percentile Query Latency (ms)

activities. Even under the same selectivity, it also varies for different queries and the standard deviation is very large.

VBASE can always produce the best query performance with high recalls since it can determine  $\tilde{K}$  on the fly.

**Q4-6 – Multi-column TopK.** Only Milvus and VBASE support Q4-6, which are TopK queries over multiple vector indices. Q5,6 adds scalar filtering to Q4 just like in Q2,3, and Milvus cannot support string filter conditions in Q6. Milvus tries different  $K'$  to produce a sufficiently large intersection of multiple TopK results from different indices. Milvus’s performance is worse than PostgreSQL which uses sequential table scan, because it cannot finish after several rounds of TopK guesses and accumulates a large number of random reads. In comparison, VBASE determines the optimal  $\tilde{K}$  per each vector index based on relaxed monotonicity. Consequently, VBASE outperforms Milvus by 200 – 300× in terms of query latencies for Q4-6, and produces higher recalls (96%+).

We also experimented multi-column TopK queries with 4 kinds of weights in the ranking function, with different index-iteration algorithms as introduced in §4.4 (see Table 7). When the difference in weights is large (1:10), the greedy algorithm produces the best performance with high recalls. This is because the greedy approach identifies low-quality indices (i.e. ones with low-ranking weights) quickly, and avoids traversing them as much as possible. However, when weight differentiation decreases, the greedy algorithm can easily get trapped in local optima. This shortcoming of the greedy method is self-evident for a weight ratio of 1:1, where we can see greedy strategy extracts the highest number of entities while producing the lowest recall. In contrast, VBASE shows higher recalls in all situations by dynamically determining a better strategy

to switch among different indices while outperforming by 5% lower latency than the round-robin approach.

**Q7 – Vector range filter.** Table 4 shows that only VBASE supports Q7. PASE does not support Q7 by default. We add a Order By distance clause with a hand-tuned “limit  $K$ ” to force PASE to use its approximate vector index. This way it simulates the results of Q7. Like in Q2, it is difficult to set an appropriate  $K'$  for PASE ahead of time as shown in Table 6. We also present the average  $\tilde{K}$  and standard deviation, which also shows  $\tilde{K}$  changes dramatically for different queries. E.g., sometimes  $K$  is required to be 2300+ for optimality. VBASE can achieve a great trade-off between query latency and recalls because its execution engine can determine  $\tilde{K}$  on-the-fly based on relaxed monotonicity.

On average Q7 only returns a small number of results that fit within the range. However, a small percentage of Q7s produce up to 10,000 results, which incurs high query costs in VBASE. Therefore we can see that, in VBASE, Q7’s 99th percentile latency is much higher than the average (168.9ms vs 10.8ms) while the median is much smaller than the average.

**Q8 – Join.** PostgreSQL performs nested-loop join on table scan to get accurate results. In Q8, VBASE is 7,900x faster with recall=0.9992. Other systems cannot run this query due to the lack of a unified query engine.

## 5.4 VBASE with SPANN

Table 8 shows the evaluation results for VBASE with SPANN [25]. We run VBASE with SPANN on Azure VM Standard\_L16s\_v3 with NVMe disks. SPANN is a partition-based ANNS index that uses external memory, i.e. disks. As

Table 5: Vector Search with Scalar Filter (Latency: ms)

System	Selectivity = 0.03 $avg(\tilde{K}) = 1,772, \sigma = 224.16$				Selectivity = 0.3 $avg(\tilde{K}) = 291, \sigma = 59.65$				Selectivity = 0.9 $avg(\tilde{K}) = 188, \sigma = 50.33$			
	Recall	Latency			Recall	Latency			Recall	Latency		
		average	median	99th		average	median	99th		average	median	99th
PASE( $K' = 100$ )	0.0567	5.1	5.1	6.3	0.5844	5.9	5.9	9.1	0.9947	5.5	5.7	10.4
PASE( $K' = 1,000$ )	0.5885	21.5	21.2	30.8	0.9998	15.4	15.0	21.5	1	10.1	10.0	16.3
PASE( $K' = 10,000$ )	1	62.8	62.5	78.7	1	48.9	49.3	61.1	1	41.8	41.7	53.1
VBASE	0.9987	34.5	34.0	44.8	0.9966	7.6	7.0	8.5	0.9990	5.7	5.3	7.2

Table 6: Range Filter (Latency: ms)

System	$avg(\tilde{K}) = 590, \sigma = 1758.48$			
	Recall	Latency		
		average	median	99th
PASE( $K' = 100$ )	0.7103	7.3	6.9	8.8
PASE( $K' = 1,000$ )	0.9387	44.3	43.6	54.7
PASE( $K' = 10,000$ )	0.9991	392.1	390.5	484.9
VBASE	0.9840	10.8	2.2	168.9

Table 7: Multi-Column TopK Comparison (Latency: ms)

Weight <sup>1</sup>	Algorithm	NumOfScans <sup>2</sup>	Latency	Recall
1 : 1	Round-Robin	651.93	20.91	<b>0.9715</b>
	Greedy <sup>3</sup>	699.02	21.70	0.9313
	VBASE	<b>638.56</b>	<b>20.56</b>	0.9705
1 : 2	Round-Robin	617.22	20.25	0.9802
	Greedy <sup>3</sup>	612.51	19.94	0.9655
	VBASE	<b>593.99</b>	<b>19.78</b>	<b>0.9818</b>
1 : 5	Round-Robin	463.39	16.93	0.9946
	Greedy <sup>3</sup>	<b>372.96</b>	<b>14.90</b>	0.9949
	VBASE	409.31	15.69	<b>0.9961</b>
1 : 10	Round-Robin	363.47	14.81	0.9981
	Greedy <sup>3</sup>	<b>274.86</b>	<b>12.69</b>	0.9985
	VBASE	311.66	13.97	<b>0.9987</b>

<sup>1</sup> The weight ratio of two distances (1 : x) in the ranking function.

<sup>2</sup> The average number of times we scan the two vector indices.

<sup>3</sup> In the greedy method, we first traverse each index 20 times and we find the index with the lowest average distance. Then, we extract candidates from this index only.

a result, query latencies on VBase with SPANN are generally higher than those for HNSW which are in-memory. This shows that VBASE can support both partition-based vector indices as well as graph-based ones. In addition VBASE can integrate indices stored both in memory and on disk seamlessly.

## 5.5 Cost Estimation

**Selectivity estimation accuracy.** We evaluate the accuracy of the selectivity estimation for vector range filter in terms of q-error [61]:

$$Q_{err} = \max\left(\frac{Sel_{esti}}{Sel_{real}}, \frac{Sel_{real}}{Sel_{esti}}\right).$$

As demonstrated in Figure 7, estimation based on sampling can provide a q-error less than 1.1 for most cases. When selectivity is lowest at 0.05, our samples cannot provide a high

Table 8: Queries on VBASE with SPANN (Latency: ms)

Queries	Recall	Latency		
		average	median	99th
Q1	0.9911	9.4	9.2	11.6
Q2	0.9214	10.7	9.3	44.9
Q3	0.9847	9.7	9.4	11.8
Q4	0.9481	32.2	28.7	68.2
Q5	0.9757	87.4	55.9	519.7
Q6	0.9516	40.1	32.1	126.2
Q7	0.9923	17.8	9.3	283.5
Q8	0.9638	87,729.3	- <sup>1</sup>	- <sup>1</sup>

<sup>1</sup> We have only run one query parameter for Q8.

resolution, therefore its q-error increases up to 1.27. Increasing the sampling rate in cost estimation can reduce q-error further, but this increases selectivity estimation time for query planning, which is infeasible for online queries. In our experiments, such estimation accuracy is sufficient to support a good query plan strategy.

In comparison, systems like PASE [11] don't provide an estimation for selectivity and PASE sets the default value of selectivity estimation to 0.5.

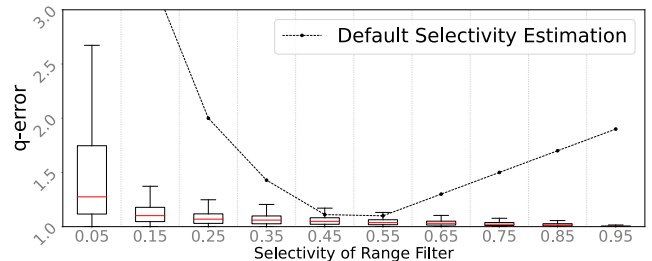


Figure 7: Q-error in different selectivity. Default estimated selectivity=0.5

**Query Planning.** We evaluate the efficacy of query planning using the following query.

```
SELECT recipe_id FROM Recipe WHERE
  → INNER_PRODUCT(q, images_embedding) < ${r}
  → AND popularity < ${p};
```

The vector range filter and the scalar filter can be accelerated via vector index or B-tree. Our experiments show that VBASE can correctly choose the best execution plan under different selectivities, because our estimations of selectivity, vector computation, and index scan cost are accurate enough for



VBASE to construct good plans by reusing PostgreSQL’s built-in mechanism.

Figure 8a shows execution times of different planning strategies for varying scalar selectivities and a fixed vector filter selectivity. The default strategy estimates selectivity as 0.5 as PASE does. The result demonstrates that VBASE can produce execution plans that closely match the ground truth of the best choice of index scan strategies. When scalar selectivity is less than 0.18, VBASE predicts accurately that the cost of execution via B-tree index is smaller than vector index traversal, and chooses to run it. Likewise, if scalar selectivity is over 0.18, VBASE correctly chooses vector index traversal.

On the other hand, experiments in Figure 8b vary vector filter selectivities and fix scalar filter selectivity. Like in Figure 8a, the result shows VBASE can create the high efficacy of query planning based on highly accurate cost estimation.

In comparison, systems like PASE do not provide an accurate estimation for selectivity. And they do not tune cost estimation for vector computation and vector index traversal either. The inaccurate estimation causes PASE to always execute queries via B-tree index in this experiment.

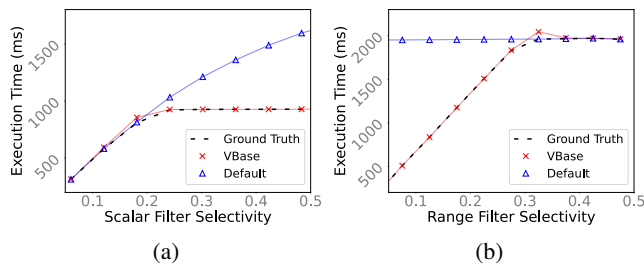


Figure 8: Query execution time with different estimation. We fixed range filter selectivity=0.13 in (a), and scalar filter selectivity=0.90 in (b)

## 6 Related Works

**Similarity Query in Databases.** Several works [20, 21, 71, 73, 74] have explored extending database systems to support accurate similarity query of low-dimensional vector data, in which  $K$ -NN( $TopK$ ) and *range filter* query are well described. R-Tree [38], KD-Tree [34], M-Tree [27], Slim-Trees [47] can be used in these works as indices for low-dimensional data. [20] proposes to include similarity queries to SQL and run them on SIREN [21]. [71] presents similarity Join and similarity Group-by operators. The similarity Group-by operator for high-dimensional vector is actually equivalent to the clustering operation, which has been well-studied by [15, 18, 50, 65]. However, all of these works are about clustering data on the main table instead of the vector indices.

**Vector Indices.** Vector indices support approximate nearest neighbor search efficiently on high-dimensional vector by  $TopK$  interface. They can be divided into two categories: graph-based approach and partition-based approach. The

partition-based approach divides the whole vector space into many sub-spaces, and uses some metric (e.g. a centroid, a hash value or a divisional plane) to represent all vectors that belong to a sub-space. During the traverse, it navigates a query to its approximate nearest sub-spaces step by step based on distances between the query and the representative metric of sub-spaces. Representative partition-based approaches include clustering-based solutions [5, 17, 19, 25, 44, 45, 48, 90], hash-based solutions [30, 41, 79, 81, 85], and tree-based solutions [23, 57, 62, 78]. The graph-based approach represents each vector as a vertex, each connected to its nearest vectors (i.e. neighbors) by edges in a graph. There are also some shortcut edges connecting to distant vector vertices, which can speed up the graph traversals. Using this neighborhood graph, index traversal can be guided by a query approximately towards its closest neighbors step by step from a fixed starting point [32, 39, 43, 55, 58, 77].  $TopK$  interface in vector indices has limited query expressiveness.

**Vector Databases based on  $TopK$ .** AnalyticDB-V [80], PASE [86], Milvus [76], and Elasticsearch [4] support complex vector queries based on the original  $TopK$  interface in vector indices. AnalyticDB-V [80] and PASE [86] integrate vector indices into the database engine to support SQL interface for similarity queries. Elasticsearch [4] is a distributed full-text search engine, providing approximate nearest neighbor search based on HNSW [89]. AnalyticDB-V [80], PASE [86], and Elasticsearch [4] begin to support vector search plus scalar attribute filtering. Milvus [76] is a data management system to efficiently manage large-scale vector data, which can additionally support multi-column  $TopK$  queries by iteratively speculating the  $K$  with a growing value. In contrast, VBASE does not rely on a tentative index collected by  $TopK$ .

## 7 Conclusion

This paper presents VBASE, a vector database that integrates high-dimensional vector indices into PostgreSQL, a relational database to facilitate complex approximate similarity queries. Unlike conventional approaches that leverage  $TopK$  to collect the target vector’s  $K$  nearest neighbors where a conventional index is constructed for query execution, VBASE builds on relaxed monotonicity, a common foundation between conventional and high-dimensional indices. This common foundation allows VBASE to build a unified query execution engine that produces query results equivalent to those produced by  $TopK$ -based solutions with the optimal  $\bar{K}$ . As a result, VBASE significantly outperforms state-of-the-art vector systems on complex vector queries.

## 8 Acknowledgement

We would like to thank our shepherd Marco Serafini and the anonymous reviewers for their insightful comments.

## References

- [1] Azure vm fsv2-series. <https://learn.microsoft.com/en-us/azure/virtual-machines/fsv2-series>.
- [2] Benchmarking nearest neighbors. <http://ann-benchmarks.com/>.
- [3] Billion-scale anns benchmarks. <https://big-ann-benchmarks.com/>.
- [4] Elasticsearch. <https://www.elastic.co/>.
- [5] Facebook faiss. <https://github.com/facebookresearch/faiss>.
- [6] Facebook simsearchnet. <https://ai.facebook.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content/>.
- [7] Google multisearch. <https://blog.google/products/search/multisearch/>.
- [8] Milvus. <https://github.com/milvus-io/milvus>.
- [9] Open distro. <https://github.com/opedistro-for-elasticsearch/>.
- [10] Openai chatgpt retrieval plugin. <https://github.com/openai/chatgpt-retrieval-plugin>.
- [11] Pase. <https://github.com/forrest-2007/PASE>.
- [12] postgresql. <https://www.postgresql.org/>.
- [13] The TPC-C benchmark. <http://www.tpc.org/tpcc/>.
- [14] The TPC-H benchmark. <http://www.tpc.org/tpch/>.
- [15] Saurabh Arora and Inderveer Chana. A survey of clustering techniques for big data analysis. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 59–65, 2014.
- [16] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, jun 1976.
- [17] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, 37(6):1247–1260, 2014.
- [18] B. Hari Babu, N. Subhash Chandra, and T. V. Gopal. Clustering algorithms for high dimensional data – a survey of issues and existing approaches. 2012.
- [19] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 202–216, 2018.
- [20] M. C. N. Barioni, H. L. Razente, A. J. M. Traina, and C. Traina. Seamlessly integrating similarity queries in sql. *Softw. Pract. Exper.*, 39(4):355–384, mar 2009.
- [21] Maria Camila N. Barioni, Humberto Razente, Agma Traina, and Caetano Traina. Siren: A similarity retrieval engine for complex data. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, page 1155–1158. VLDB Endowment, 2006.
- [22] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.
- [23] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [24] Donald D. Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012.
- [25] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 5199–5212. Curran Associates, Inc., 2021.
- [26] Sheng Chen, Akshay Soni, Aasish Pappu, and Yashar Mehdad. Doctag2vec: An embedding based multi-label learning approach for document tagging. *arXiv preprint arXiv:1707.04596*, 2017.
- [27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. *International conference on very large data bases (VLDB)*, 08 2001.
- [28] Kenneth L Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 160–164, 1994.
- [29] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.
- [30] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme

- based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, 2004.
- [31] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
- [32] Wei Dong, Moses Charikar, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 577–586, 2011.
- [33] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In Peter Buneman, editor, *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.
- [34] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, sep 1977.
- [35] G. Graefe. Volcano an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, feb 1994.
- [36] Wayne D Gray and Deborah A Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of experimental psychology: applied*, 6(4):322, 2000.
- [37] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, 2022.
- [38] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84*, page 47–57, New York, NY, USA, 1984. Association for Computing Machinery.
- [39] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1312–1317, 2011.
- [40] G. D. Held, M. R. Stonebraker, and E. Wong. Ingres: A relational data base system. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition, AFIPS '75*, page 409–416, New York, NY, USA, 1975. Association for Computing Machinery.
- [41] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.
- [42] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.
- [43] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.
- [44] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.
- [45] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.
- [46] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547, 2021.
- [47] Caetano Jr, Agma Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. 03 2000.
- [48] Yannis Kalantidis and Yannis Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2321–2328, 2014.
- [49] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 535–544, 2012.
- [50] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1), mar 2009.

- [51] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2130–2137, 2009.
- [52] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.
- [53] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. Fexipro: fast and exact inner product retrieval in recommender systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 835–850, 2017.
- [54] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. The design and implementation of a real time visual search system on JD e-commerce platform. In *Proceedings of the 19th International Middleware Conference, Middleware Industrial Track 2018, Rennes, France, December 10-14, 2018*, pages 9–16. ACM, 2018.
- [55] Jie Ren, Minjia Zhang, Dong Li. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. In *Advances in Neural Information Processing Systems*, 2020.
- [56] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. Lightrec: A memory and search-efficient recommender system. In *Proceedings of The Web Conference 2020*, pages 695–705, 2020.
- [57] Ting Liu, Andrew W Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, {NIPS} 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pages 825–832, 2004.
- [58] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.
- [59] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m+: A dataset for learning cross-modal embeddings for cooking recipes and food images. *IEEE transactions on pattern analysis and machine intelligence*, 43(1):187–203, 2019.
- [60] Antoine Miech, Dimitri Zhukov, Jean-Baptiste Alayrac, Makarand Tapaswi, Ivan Laptev, and Josef Sivic. Howto100m: Learning a text-video embedding by watching hundred million narrated video clips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2630–2640, 2019.
- [61] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.*, 2(1):982–993, aug 2009.
- [62] Marius Muja and David G. Lowe. Scalable Nearest Neighbour Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.
- [63] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. Ms marco: A human generated machine reading comprehension dataset. In *CoCo@ NIPS*, 2016.
- [64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.
- [65] Divya Pandove, Shivan Goel, and Rinki Rani. Systematic review of clustering high-dimensional and large datasets. *ACM Trans. Knowl. Discov. Data*, 12(2), jan 2018.
- [66] Mattis Paulin, Matthijs Douze, Zaid Harchaoui, Julien Mairal, Florent Perronin, and Cordelia Schmid. Local convolutional features with unsupervised training for image retrieval. In *Proceedings of the IEEE international conference on computer vision*, pages 91–99, 2015.
- [67] Jianbin Qin, Wei Wang, Chuan Xiao, and Ying Zhang. Similarity query processing for high-dimensional data. *Proc. VLDB Endow.*, 13(12):3437–3440, sep 2020.
- [68] Amaia Salvador, Nicholas Hynes, Yusuf Aytar, Javier Marin, Ferda Ofli, Ingmar Weber, and Antonio Torralba. Learning cross-modal embeddings for cooking recipes and food images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3020–3028, 2017.
- [69] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.



- [70] Minjoon Seo, Jinhyuk Lee, Tom Kwiatkowski, Ankur P Parikh, Ali Farhadi, and Hannaneh Hajishirzi. Real-time open-domain question answering with dense-sparse phrase index. *arXiv preprint arXiv:1906.05807*, 2019.
- [71] Yasin N. Silva, Walid G. Aref, Per-Ake Larson, Spencer S. Pearson, and Mohamed H. Ali. Similarity queries: Their conceptual evaluation, transformations, and processing. *The VLDB Journal*, 22(3):395–420, jun 2013.
- [72] Yukihiro Tagami. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 455–464, 2017.
- [73] Caetano Traina, Andre Moriyama, Guilherme Rocha, Robson Cordeiro, Cristina D. A. Ciferri, and Aagma Traina. The similarql framework: Similarity queries in plain sql. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19*, page 468–471, New York, NY, USA, 2019. Association for Computing Machinery.
- [74] Caetano Traina, Aagma J. M. Traina, Marcos R. Vieira, Adriano S. Arantes, and Christos Faloutsos. Efficient processing of complex similarity queries in rdbms through query rewriting. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management, CIKM '06*, page 4–13, New York, NY, USA, 2006. Association for Computing Machinery.
- [75] Robert E. Wagner. Indexing design considerations. *IBM Systems Journal*, 12(4):351–367, 1973.
- [76] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.
- [77] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113. IEEE, 2012.
- [78] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian Sheng Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(2):388–403, 2014.
- [79] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, 2018.
- [80] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdbv: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.
- [81] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *Advances in neural information processing systems*, pages 1753–1760, 2009.
- [82] Xian Wu, Moses Charikar, and Vishnu Natchu. Local density estimation in high dimensions. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5296–5305. PMLR, 10–15 Jul 2018.
- [83] Xiang Wu, Ruiqi Guo, David Simcha, Dave Dopson, and Sanjiv Kumar. Efficient inner product approximation in hybrid spaces. *ArXiv*, abs/1903.08690, 2019.
- [84] Shitao Xiao, Zheng Liu, Weihao Han, Jianjin Zhang, Yingxia Shao, Defu Lian, Chaozhuo Li, Hao Sun, Denvy Deng, Liangjie Zhang, et al. Progressively optimized bi-granular document representation for scalable embedding based retrieval. In *Proceedings of the ACM Web Conference 2022*, pages 286–296, 2022.
- [85] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. Complementary hashing for approximate nearest neighbor search. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1631–1638. IEEE, 2011.
- [86] Wen Yang, Tao Li, Gai Fang, and Hong Wei. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2241–2253, 2020.
- [87] Ian En-Hsu Yen, Satyen Kale, Felix Yu, Daniel Holtmann-Rice, Sanjiv Kumar, and Pradeep Ravikumar. Loss decomposition for fast learning in large output spaces. In *International Conference on Machine Learning*, pages 5640–5649. PMLR, 2018.
- [88] Zeynep Akkalyoncu Yilmaz, Shengjin Wang, Wei Yang, Haotian Zhang, and Jimmy Lin. Applying bert to document retrieval with birch. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, pages 19–24, 2019.

- [89] Malkov, D A Yashunin, Yu A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 824–836, 2018.
- [90] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *Proceedings of the 31th International Conference on Machine Learning (ICML)*, volume 32, pages 838–846, 2014.





# Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction

Zu-Ming Jiang  
ETH Zurich

Si Liu  
ETH Zurich

Manuel Rigger  
National University of Singapore

Zhendong Su  
ETH Zurich

## Abstract

Transactions are an important feature of database management systems (DBMSs), as they provide the ACID guarantees for a sequence of database operations. Consequently, approaches have been proposed to automatically find transactional bugs in DBMSs. However, they cannot handle complex operations and predicates common in real-world database queries, and thus miss bugs.

This paper introduces a general, effective technique for finding transactional bugs in DBMSs that supports complex SQL queries and predicates. At the conceptual level, we address the test-oracle problem by constructing semantically-equivalent test cases based on fine-grained statement-level dependencies in transactions. At the technical level, we introduce (1) *statement-dependency graphs* to describe dependencies among SQL statements in transactions, (2) *SQL-level instrumentation* to capture possible statement-level dependencies, and (3) *transactional oracle construction* to generate semantically-equivalent test cases using statement-dependency graphs. We also establish the correctness of our approach in generating semantically-equivalent test cases. We have realized our technique as a tool, TxCheck, and evaluated it on three widely-used and well-tested DBMSs, namely TiDB, MySQL, and MariaDB. In total, TxCheck found 56 unique bugs, 52 of which have been confirmed and 18 already fixed. We believe that TxCheck can help solidify DBMSs' support for transactions thanks to its generality and effectiveness.

## 1 Introduction

Database management systems (DBMSs) store and manage data and are crucial for many applications. A key feature of DBMSs is their support for transactions, where a sequence of SQL statements are executed as a single unit, and various properties (*i.e.*, atomicity, consistency, isolation, and durability) are guaranteed. For example, if some transactions are concurrently executed at the *Serializability* isolation level, uncommitted operations by other transactions will be invisible, and the transaction execution results must be equal to

the results when these transactions are executed in a serial order. Benefiting from these properties, transactions have been applied in many critical applications. However, transaction implementations usually involve complex logic (*e.g.*, two-phase locking [7, 45] and multiversion concurrency control [8, 32]), and thus bugs are easily introduced. In this paper, we refer to the bugs in the transaction support of DBMSs as *transactional bugs*. Such bugs are critical because they can paralyze their client applications or, even worse, silently trigger incorrect behaviors in critical operations of client applications.

To improve the reliability and correctness of transaction processing in DBMSs, several approaches [4, 9, 11, 17, 44] have been proposed to test transaction support. These approaches use specific operation patterns to capture the violations of transactional rules. For example, ELLE [4] encodes transaction execution histories<sup>1</sup> by only appending to a conceptual list data structure. It builds transaction-dependency graphs based on the histories, and reports bugs if the graphs violate the desired isolation guarantees. Limited by specific test-case patterns, these approaches use only simple operations (*e.g.*, ELLE appends list with constant values), while many deep bugs may only be triggered by complex operations [21]. Moreover, existing approaches [4, 9, 17, 44] cannot handle predicates (*e.g.*, the condition expressions in **WHERE** clauses) in general. For example, ELLE cannot encode predicate operations in its list data structure. However, predicates are ubiquitous in real-world transactions as they rely on common features such as **WHERE** clauses or **JOINS**. Their lack of predicate support makes existing approaches miss many real bugs. In addition, transactional bugs may be independent of isolation levels (*e.g.*, incorrect results returned by one transaction), while existing approaches focus on testing isolation levels, thus missing bugs.

Figure 1(a) shows a confirmed bug in MySQL at the *Repeatable Read* isolation level. The bug-triggering test case involves two tables and two interleaved transactions. The statement T1.S3 is executed immediately after T1.S2. The

<sup>1</sup>A history records transactional requests to and responses from a database.



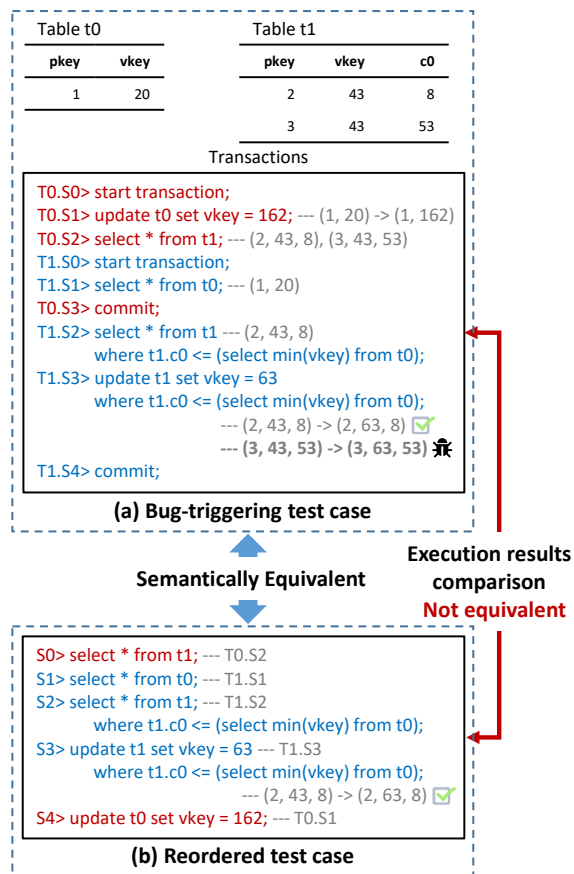


Figure 1: A MySQL bug found by TxCheck under the *Repeatable Read* isolation level.

oretically,  $T1.S3$  should fetch exactly those records subsequently updated by  $T1.S2$ , because they use the same predicate (*i.e.*, the same expression in their **WHERE** clause), and no other operations are executed between them. However,  $T1.S2$  fetches only the row (2, 43, 8), while  $T1.S3$  updates rows (2, 43, 8) and (2, 43, 53) due to its incorrect predicate matching. Existing approaches cannot find this bug for two reasons. First, the test case uses an aggregate function (*i.e.*, `min()`) and a subquery (*i.e.*, the **SELECT** in the **UPDATE** statement), which make the test case complex and not follow the test-case patterns of existing approaches (*e.g.*, ELLE can append only constant values instead of `min()` values). Second, the test case uses predicates, for which existing approaches lack support.

Figure 1(b) shows a test case generated by our approach, which is equivalent to the one shown in Figure 1(a). The three **SELECT** statements (*i.e.*,  $S0$ ,  $S1$ , and  $S2$ ) are moved before the two **UPDATE** statements (*i.e.*,  $S3$  and  $S4$ ), because all these **SELECT** statements are oblivious to the effects of the **UPDATE** statements. As  $T1.S3$  is executed immediately after  $T1.S2$ , the database state visible for  $T1.S2$  and  $T1.S3$  should be consistent. Therefore, we keep the statements of  $T1.S2$  and  $T1.S3$  adjacent (*i.e.*,  $S2$  and  $S3$ ). The reordered test case is executed *without using transactions*, and its execution results

should be the same as the original one, because the reordering does not change any expected behavior of each statement. In this case, the bug in MySQL breaks the equivalence.

Our insight is to generate semantically-equivalent test cases that are not wrapped as transactions, but produce the same execution results, by properly reordering the statements. Then, we can validate whether their equivalence indeed holds. Any discrepancy indicates a bug in the target DBMS. While for the aforementioned test case, it is intuitive that reordering the statements will not affect the execution results of follow-up test cases, we must reason, in general, about the dependencies of statements. To this end, we first propose *statement-dependency graphs*, a novel concept to describe the dependencies among executed statements, which provide finer-grained dependency information than transaction-dependency graphs [1, 17, 44]. This facilitates finding more bugs (as will be discussed in Section 5.3). To extract statement dependencies, we propose *SQL-level instrumentation*. Specifically, we insert additional statements to collect the execution results of each target statement in transactions. Based on the collected results, we can track the operation effects—including the effects of predicate operations—of each statement, and thus infer all possible statement dependencies. To generate semantically-equivalent test cases, we propose *transactional oracle construction*. Specifically, we topologically sort the acyclic statement-dependency graphs, whose sorted statement sequences are proved to be semantically equivalent to the original one. To guarantee the acyclicity of graphs, we iteratively delete statements in cycles before sorting. We execute the sorted statement sequences without transactions and compare their results to those from the corresponding transaction executions. Any difference reveals a bug in the tested DBMS.

We realized this approach as a practical tool called TxCheck. We evaluate TxCheck on three popular and extensively tested DBMSs, namely TiDB [46], MySQL [30], and MariaDB [29]. In total, TxCheck found 56 unique bugs, including 23 in TiDB, 18 in MySQL, and 15 in MariaDB. Among them, 52 bugs have been confirmed, 18 fixed, and 8 assigned CVE IDs; 30 are triggered in transaction executions. These results collectively demonstrate that TxCheck can find latent transactional bugs in mature production DBMSs.

Overall, we make the following contributions:

- At the conceptual level, we address the test-oracle problem of DBMS transaction testing by constructing semantically-equivalent test cases.
- At the technical level, we propose (1) statement-dependency graphs, which describe the dependencies among statements in executed transactions, (2) SQL-level instrumentation, which can capture all possible statement dependencies including the predicate-related dependencies, and (3) transactional oracle construction, which refines test cases and generates semantically-equivalent test cases for validation. We formally prove the correctness of our approach.

- At the practical level, we implement our approach into a tool, TxCheck, and evaluate it on three widely-used DBMSs (*i.e.*, TiDB, MySQL, and MariaDB). In total, TxCheck finds 56 unique bugs, most of which cannot be identified by existing approaches. TxCheck is open-sourced at <https://github.com/JZuming/TxCheck>.

## 2 Background

**Transactions in DBMSs.** A database transaction refers to a series of operations, for which DBMSs must guarantee atomicity, consistency, isolation, and durability (*i.e.*, *ACID*) [49]. This paper focuses on relational database management systems (RDBMSs), where a transaction typically consists of a group of SQL (*i.e.*, Structured Query Language) statements. Each statement performs read operations (*e.g.*, **SELECT** statements) or write operations (*e.g.*, **INSERT**, **DELETE**, and **UPDATE** statements). SQL statements commonly involve predicates (*e.g.*, **WHERE** clauses) for choosing desired rows that satisfy the requirements.

**Dependency Graphs.** Adya *et al.* [1, 2] propose transaction dependencies, which can be classified into two categories, namely *item dependencies* and *predicate dependencies*. Item dependencies describe the relationship among transactions on specific items (*i.e.*, rows in tables) they read from or write to. For example, transaction  $T_j$  *item-read-depends* on  $T_i$  if  $T_j$  reads an item version  $x_i$  that is written by  $T_i$ . A predicate dependency describes the relationship between two transactions constructed from the associated predicate operations. For example, transaction  $T_j$  *directly predicate-read-depends* on  $T_i$  if an item version  $x_i$  that is written by  $T_i$  is used for predicate matching of  $T_j$ .

Based on these dependencies, Adya *et al.* propose a transaction-dependency graph, called *Direct Serialization Graph* (DSG). DSGs can be used to formalize the expected behaviors of DBMSs under different isolation levels. For example, *Serializability* (PL-3) proscribes any directed cycle in a DSG, while *Repeatable Read* (PL-2.99) any directed cycles that dismiss certain predicate dependencies. Bailis *et al.* [5, 6] further extend DSGs to define several other isolation levels.

**Existing Approaches.** Both transaction-focused testing and verification approaches [4, 9, 11, 17, 44] have utilized dependency graphs. They typically use specific operation patterns to capture transaction dependencies. For example, to reduce the search space of possible transaction orders, COBRA [44] exploits the read-modify-write (RMW) patterns where a transaction reads from a key before writing to it. By restricting its writes to list-specific operations like “append”, ELLE [4] can naturally infer the transaction order from a list of values read.

While existing approaches [4, 9, 17, 44] such as ELLE have been successful in detecting a wide range of important bugs, they are limited in finding *deep* transactional bugs for two

main reasons. First, these approaches can use only simple operations (*e.g.*, writing key-value pairs) following their restricted operation patterns. However, many deep DBMS bugs can only be triggered by complex operations [21]. Second, existing approaches lack support for predicate operations in general. In contrast to read/write operations whose effects are explicitly reflected in the final execution results, the effects of predicate operations are implicit and difficult to track, because they are typically reflected in the intermediate processes (*e.g.*, choosing *a set of items* that satisfy the predicate conditions for subsequent read/write operations). However, predicates are commonly used in real-world transactions and involved in sophisticated features of DBMSs like predicate optimization [23]. Existing approaches do not consider predicate operations in their test cases, thus missing many bugs with respect to the transaction support of DBMSs. In addition, existing approaches focus on testing isolation guarantees while many transactional bugs are not necessarily related to database isolation.

## 3 Approach

In this section, we present a novel approach for addressing the challenges of testing transaction support in DBMSs, as illustrated in Figure 2. Our core idea is to extract the dependencies among statements in transactions and construct semantically-equivalent test cases according to the extracted dependency information. The constructed test cases are used as oracles to validate the original transaction executions by checking whether all statements in the test cases produce the same results. To realize this idea, we first define statement dependencies and propose a new concept, *statement-dependency graphs*, which describes the dependencies among statements in transactions. To derive statement-dependency graphs, we capture dependency information from specific transaction executions. Then, we generate semantically-equivalent test cases based on the captured information.

**Dependency Capturing.** To capture statement dependencies, we propose *SQL-level instrumentation*, which inserts SQL statements to collect execution information of each original statement in transactions. Using SQL to achieve this makes this approach a *black-box technique* that is applicable even for DBMS where testers lack access to the source code. Specifically, we instrument in two steps, item-tracking instrumentation and version-set-tracking instrumentation, which capture item dependencies and predicate dependencies, respectively. Statement-dependency graphs are built based on the outputs from the inserted and the original statements.

**Oracle Construction.** We propose *transactional oracle construction* to construct semantically-equivalent test cases. We first iteratively remove statements involved in cycles to make the statement-dependency graph acyclic, which is the precondition of the construction. Then, we perform topological sort-

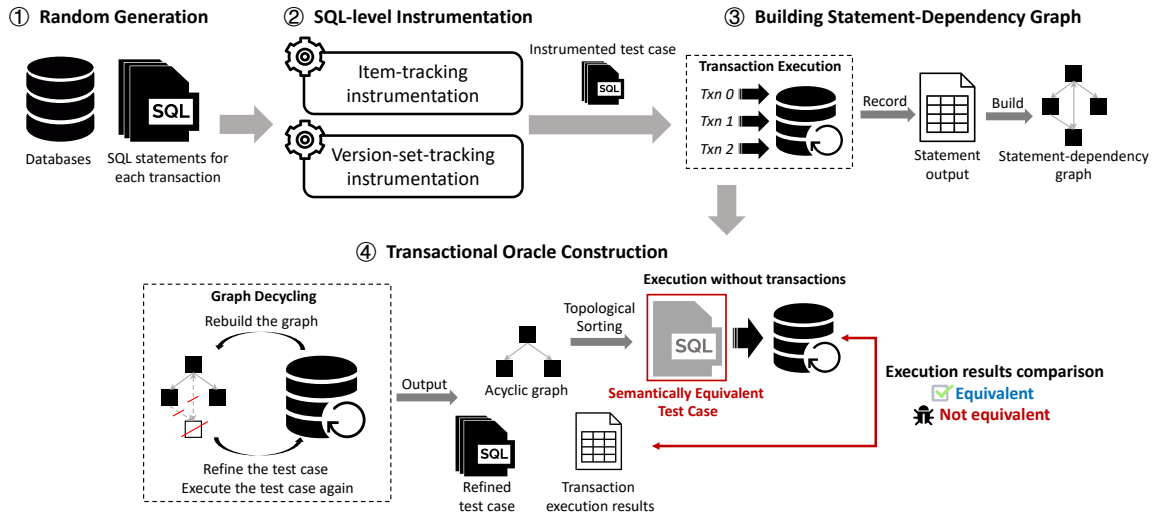


Figure 2: Approach overview.

ing on the acyclic graph to construct semantically-equivalent test cases. We execute these test cases without transactions and compare their execution results to those from the corresponding transaction executions. We prove that, for any correct DBMS, these results should be identical, thus any difference indicates an actual DBMS bug.

### 3.1 Statement-Dependency Graph

To construct semantically-equivalent test cases by reordering statements, we need to identify the dependencies between statements. We define seven kinds of statement dependencies, referring to transaction dependencies defined by Adya *et al.* [1, 2]. Each kind of statement dependency is shown in Figure 3. Specifically, we define three kinds of statement dependencies, shown in Figure 3(a)–(c), to model the relationship of two statements that read or write the same items. Three other kinds of dependencies, shown in Figure 3(d)–(f), are used to represent the dependencies established by predicate operations. The last one, direct stmt-value-write dependency shown in Figure 3(g), is used to model the event that the new value of an item installed by a statement is determined by the value of another item that is installed by another statement. The formal definitions (Definition 2–8) for these statement dependencies are given in Appendix A. Note that we do not count statement orders as dependencies, because when two statements access different data, the statements’ execution order will not affect their results.

In contrast to the dependencies defined by Adya *et al.* [1, 2], which describe the relationship between transactions, our definitions model the relationship between statements to provide finer-grained dependency information. Statement dependencies enable us to analyze the effects of each statement, which is needed for generating semantically-equivalent test cases accordingly, while transaction dependencies lack sufficient

information related to statements. Next, we further propose statement-dependency graphs, *SDG*, to model the executions of test cases at the statement level.

#### Definition 1 (Statement-Dependency Graph)

We define the statement-dependency graph constructed based on a statement execution history  $H$ , denoted as  $SDG(H)$ , as follows.  $SDG(H)$  is a directed graph, whose nodes represent the statements in committed transactions, and whose (directed) edges represent the dependencies between these committed statements. In particular, if statement  $S_j$  depends on statement  $S_i$ , there is a direct edge from  $S_i$  to  $S_j$ .

Note that statement-dependency graphs consider only statements in committed transactions. The statements in aborted transactions are dismissed because these statements conceptually do not affect the manipulated databases and other committed transactions. Figure 4 shows the statement-dependency graph for the test case in Figure 1(a). Statement-dependency graphs contain all dependency information related to statements and reflect the execution results of test cases at the statement level, which is the basis for generating semantically-equivalent test cases.

### 3.2 SQL-Level Instrumentation

To build statement-dependency graphs, we extract statement dependencies from transaction executions. In contrast to existing work, we aim to support test cases whose statements use predicates, without involving too many restrictions on test-case patterns. This section presents *SQL-level instrumentation*, a novel technique for extracting statement dependencies from transaction executions involving predicates.

The basic idea of SQL-level instrumentation is to insert statements to output the handled items before and after the operations performed by target statements. To realize this

Table t (id, value): (0,0)

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> select * from t; --- (0, 1) ← is dependent on
```

(a) Direct stmt-item-read dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> update t set value = 2; --- (0, 1) -> (0,2)
is dependent on ↓
```

(b) Direct stmt-item-write dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> select * from t; --- (0, 0) ← is dependent on ↑
```

(c) Direct stmt-item-anti dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> select * from t where value = 0; --- empty
is dependent on ↓
```

(d) Direct stmt-predicate-read dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> update t set value = 2 where value = 0; --- change nothing
is dependent on ↓
```

(e) Direct stmt-predicate-write dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> select * from t where value = 1; --- empty
is dependent on ↑
```

(f) Direct stmt-predicate-anti dependency

```
Sj> update t set value = 1; --- (0,0) -> (0,1)
...
Sj> insert into t values (1, (select min(value) from t)); --- insert (1, 1)
is dependent on ↓
```

(g) Direct stmt-value-write dependency

Figure 3: Examples for each kind of statement dependency.

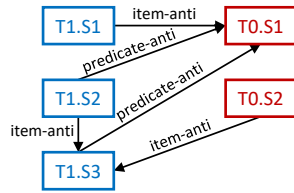


Figure 4: Statement-dependency graph for Figure 1(a).

idea, SQL-level instrumentation requires tables in manipulated databases to contain at least two columns. We name these two columns as *PrimaryKey* column and *VersionKey* column, respectively. The column *PrimaryKey* is used to identify different items, and should not change after the items are inserted. The column *VersionKey* is used to identify different versions of each item, and should be assigned a new value different from any earlier values when the item is updated by statements. Besides these two, tables may have additional columns whose properties are unrestricted.

The statements in test cases should also follow these restrictions. Specifically, the statement performing write operations (e.g., **UPDATE** and **INSERT**) should change the *VersionKey* of the handled items to a new value, and each item updated by the same statement has the same *VersionKey*. The statement performing read operations (e.g., **SELECT**) should at least output the *PrimaryKey* and *VersionKey* of the items. Except for the *PrimaryKey* and *VersionKey* of items, we eschew imposing any additional restrictions for the generated statements.

SQL-level instrumentation operates in two phases: (1) item-tracking instrumentation, and (2) version-set-tracking instrumentation. Figure 5 shows how each phase instruments the test case in Figure 1. In item-tracking instrumentation, we insert a *Before-Write Read (BWR)* statement before each statement performing write operations (e.g., see T0.S1.BWR and T1.S3.BWR in Figure 5). *BWR* statements are designed to output the items that will be written and thus use the same predicates as the target statements. *BWR* statements can work only under isolation levels that satisfy Assumption 3, which

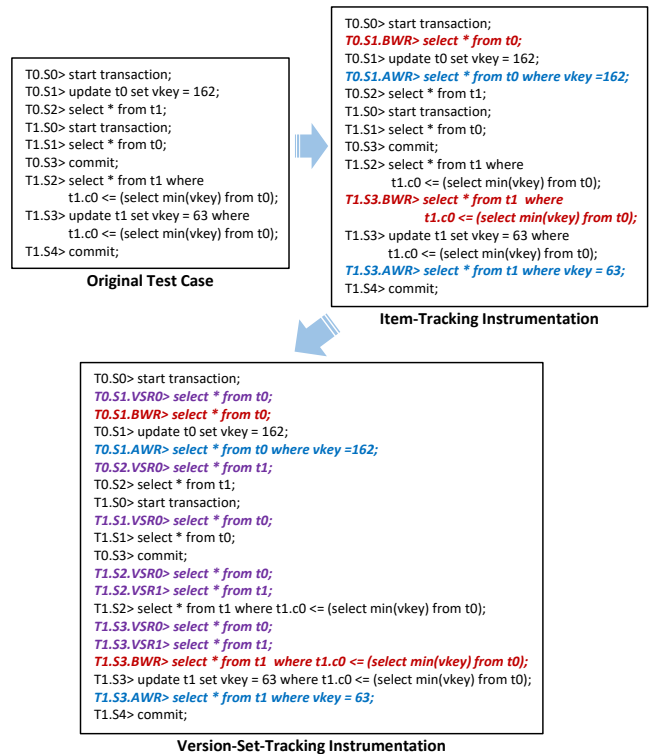


Figure 5: SQL-level instrumentation for Figure 1(a).

is discussed subsequently. We insert an *After-Write Read (AWR)* statement after each statement performing write operations (e.g., see T0.S1.AWR and T1.S3.AWR in Figure 5). *AWR* statements are used to output the new values of the items processed by target statements. To do so, *AWR* statements select items whose *VersionKeys* are equal to the assigned values in the target statements. In version-set-tracking instrumentation, we insert some *Version-Set Read (VSR)* statements before each statement (e.g., see T0.S1.VSR0 and T0.S2.VSR0 in Figure 5). To output the item versions referenced by target statements, *VSR* statements output all items in the tables ref-



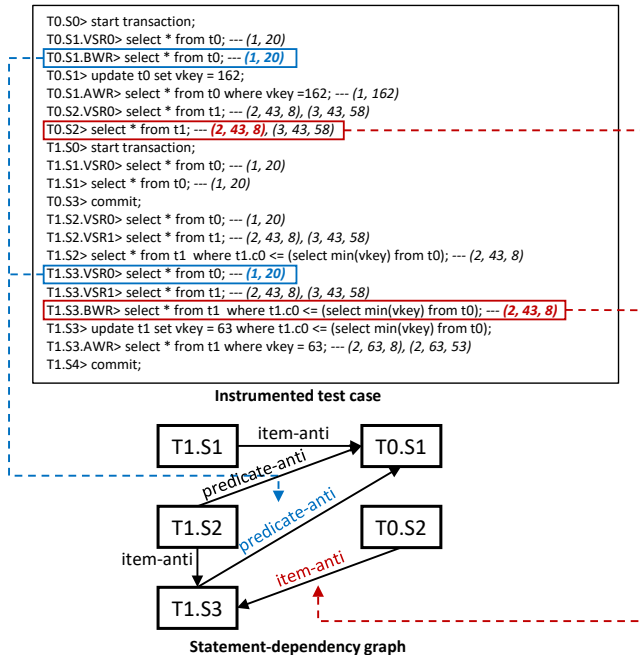


Figure 6: Inferring statement-dependency graphs for Figure 1.

erenced by the target statements.

Using the inserted *BWR*, *AWR*, and *VSR* statements, we can collect the execution information of each statement in transactions, and thus can infer possible statement dependencies. Figure 6 shows how the inserted statements are used to infer statement dependencies for the test case in Figure 1. To check whether statements  $T1.S3$  and  $T0.S2$  have dependencies between each other, we analyze the outputs of the corresponding statements and their inserted statements. The output of  $T0.S2$  and the output of *BWR* of  $T1.S3$  have an overlapping part (*i.e.*, item (2, 43, 8)), which means  $T0.S2$  reads an item that has not been updated by  $T1.S3$ . Therefore,  $T0.S2$  is (stmt-item-anti) depended on  $T1.S3$ . The outputs of *BWR* of  $T0.S1$  and *VSRs* of  $T1.S3$  also have an overlapping part (*i.e.*, item (1, 20)), which means that  $T0.S1$  will update an item that has been referenced by the predicates of  $T1.S3$ . Therefore,  $T1.S3$  is (stmt-predicate-anti) depended on  $T0.S1$ . Other dependencies can be inferred similarly.

We prove that each statement dependency proposed in Section 3.1 can be inferred based on the outputs of statements under certain assumptions. The detailed proof (Lemma 1–7) is presented in Appendix B. The assumptions are shown below:

**Assumption 1** No synchronization issues happen during the execution of transactions.

**Assumption 2** Statements can use item versions only in the tables that they have referenced.

**Assumption 3** For any two transactions,  $T_i$  and  $T_j$ , it is prohibited that  $T_i$  item-anti-depends on  $T_j$  for the item  $x$  while  $T_j$  item-write-depends on  $T_i$  for the same item  $x$ .

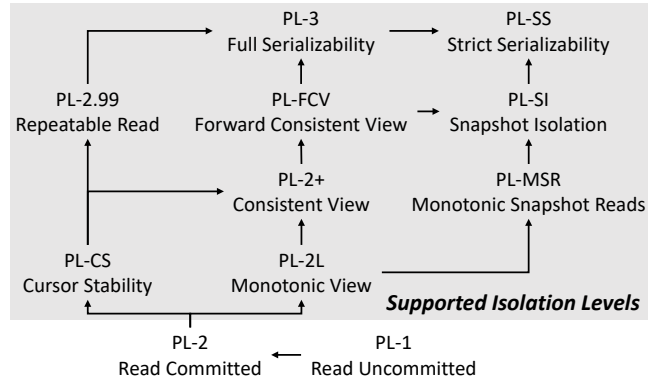


Figure 7: Isolation-level hierarchy defined by Adya *et al.* [1,2] and isolation levels supported by our approach.

These assumptions generally hold when we test DBMSs deployed locally (*i.e.*, without synchronization issues) with a proper isolation guarantee, which can be satisfied by the isolation levels equal to or stronger than Cursor Stability (PL-CS) or Monotonic View (PL-2L), according to Adya *et al.*'s definitions [1, 2]. Figure 7 shows the supported isolation levels. Specifically, Assumptions 1 and 2 are independent of isolation levels. In a transaction dependency graph, Cursor Stability prohibits cycles with an anti-dependency and one or more write-dependency edges such that all edges are related to one specific object. Assumption 3 prohibits cycles with exactly one anti-dependency edge and one write-dependency edge such that both edges are related to one specific object. Therefore, Cursor Stability satisfies Assumption 3. Monotonic View disallows cycles containing exactly one anti-dependency edge from one transaction to another transaction. It satisfies Assumption 3, because the phenomenon prohibited by Assumption 3 contains cycles with exactly one anti-dependency edge between transactions.

SQL-level instrumentation can accurately capture item dependencies without any false positives or negatives, but it may capture spurious predicate or value dependencies, according to Lemma 1–7. Such inaccuracies may be introduced by *VSR* statements; *VSR* statements output all items in the tables referenced by target statements. However, target statements may use only a part of the items in the tables to perform their predicate matching and value capturing, depending on the specific implementations of the DBMS. Therefore, *VSR* statements may output items that are not referenced by target statements. If these incorrectly outputted items match the outputs of other statements, spurious dependencies are captured. Therefore, the statement-dependency graph built by SQL-level instrumentation is a super graph of the actual statement-dependency graph. This issue does not affect the correctness of our testing approach (see Section 3.3 for the detailed discussion).

We further discuss the time complexity of using SQL-level instrumentation to infer dependencies. Suppose the database contains  $n$  items, and the test case contains  $m$  statements. In

the worst case, each *BWR* or *AWR* statement can output  $n$  items, and the *VSR* statements of each target statement can output, in total,  $n$  items. If the target statement is a read statement, it can output at most  $n$  items. When both two target statements are write statements, we need to check  $6n$  items (*i.e.*, outputs of *BWR*, *AWR*, and *VSR* statements of each target statement), while we need only check  $5n$  items when one of the target statements is a read statement (*i.e.*, outputs of *BWR*, *AWR*, and *VSR* statements of the write statement and outputs of the read statement and its *VSR* statements). Therefore, in the worst case, we may check whether two target statements have dependencies in  $O(6n)$  time using hash tables. To confirm the dependencies among  $m$  statements, we should check  $m(m - 1)$  dependencies, and thus the worst-case time complexity of the entire process is  $O(6n \cdot m(m - 1))$ , *i.e.*,  $O(m^2n)$ .

Existing work proves that checking histories in isolation levels is a polynomial-time (*e.g.*, *Read Committed*) or even NP-complete problem (*e.g.*, *Serializability* and *Snapshot Isolation*) [9, 33]. ELLE [4] can recover histories in  $O(m \cdot p)$ , where  $m$  is the number of operations, and  $p$  is the number of concurrent processes. However, ELLE restricts their test cases whose write operations can only append and cannot handle histories involving predicates. In contrast, SQL-level instrumentation can recover histories involving predicates and only requires test cases to maintain *PrimaryKey* and *VersionKey* for each item, within  $O(m^2n)$  time.

### 3.3 Transactional Oracle Construction

The statement-dependency graphs enable us to construct semantically-equivalent test cases. Our intuition is that if there is a reordered statement sequence whose statements follow the same dependency order in the statement dependency graph, the reordered statement sequence should produce the same results as the original one. To effectively test DBMSs' transaction support, we execute the reordered statements without transactions, which provides an oracle for validating the original test cases by checking whether each statement in the test cases produces the same results. To formalize our intuition, we first introduce a theorem based on statement-dependency graphs. The theorem is given below, and the details of its proof can be found in Appendix C.

**Theorem 1**  $SDG(H)$  is the statement-dependency graph built according to execution history  $H$ ,  $S(SDG(H))$  is the statement sequence generated by performing topological sorting on  $SDG(H)$ , and  $History(S(SDG(H)))$  is the history of the sorted statement sequence executed without transactions. If  $SDG(H)$  is acyclic,  $History(S(SDG(H)))$  and  $H$  give the same results for each statement in  $S(SDG(H))$ .

Theorem 1 suggests a high-level method for constructing semantically-equivalent test cases. Theorem 1 is not constrained to any specific isolation level and thus can apply at various isolation levels. Moreover, Theorem 1 can tolerate the

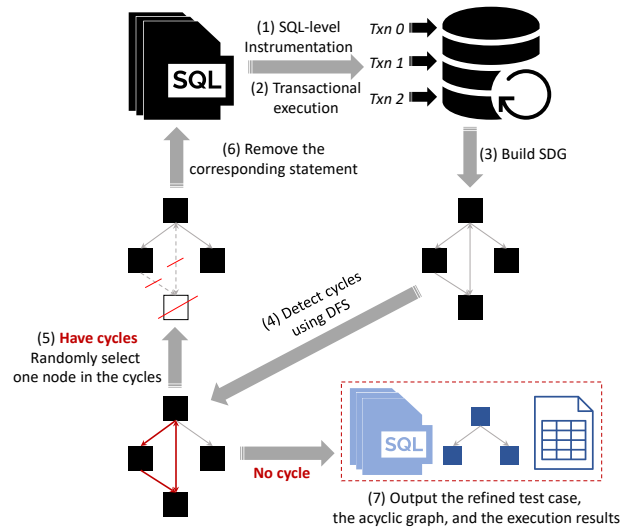


Figure 8: The process of refining test cases, whose SDG eventually becomes acyclic.

spurious predicate dependencies that stem from SQL-level instrumentation. Suppose  $G$  is the actual statement dependency graph and  $G'$  is  $G$  with additional spurious predicate dependencies, *i.e.*,  $G$  is a subgraph of  $G'$ . The semantically-equivalent test case constructed from  $G'$  will follow all the dependencies in  $G'$ , thus following all the dependencies in  $G$ . Hence, the constructed test case is also one of the topological sorting results of  $G$ . That is, the test case constructed from  $G'$  can also serve as an oracle.

Note that Theorem 1 has a precondition, *i.e.*, the  $SDG(H)$  should be acyclic. To satisfy this precondition, we perform *graph decycling* to eliminate all cycles in the graph. Then, we perform *oracle checking* to generate semantically-equivalent test cases by topologically sorting acyclic graphs. We execute these equivalent test cases without transactions and use their results to validate transaction executions.

**Graph Decycling.** The overview of graph decycling is shown in Figure 8. The idea is to break cycles in the graph by removing those statements involved in the cycles. Given a test case, we first instrument it (Step (1) in Figure 8) and then execute the instrumented test case using transactions (Step (2)). We can infer the SDG using the collected information from instrumented statements (Step (3)). Then, we check whether there is a cycle in the constructed SDG using depth-first search [39] (Step (4)). If there is at least one cycle in the graph, we randomly select one node in the cycles (Step (5)) and remove the corresponding statement in the test case (Step (6)). We re-execute the refined test case on the DBMS and start a new round of test-case refinement. Note that the re-execution is necessary, because a refined test case may result in the construction of a significantly different SDG, which might also contain new cycles. At the beginning of each round, the tested DBMS is reset to its initial state. When no cycle is detected in the SDG built from the refined test case, we output the refined

## Algorithm 1: Oracle Checking

```

input : test_case, graph, t_results
1 Function OracleChecking(test_case, graph, t_results, DBMS):
2   oracle_test_case ← OracleGen(test_case, graph);
3   DBMS ← INITIAL_STATE;
4   o_results ← ExecuteWithoutTxn(oracle_test_case, DBMS);
5   for each stmt in oracle_test_case do
6     t_stmt_results ← GetStmtResults(stmt, t_results);
7     o_stmt_results ← GetStmtResults(stmt, o_results);
8     if t_stmt_results ≠ o_stmt_results then
9       ReportBug();
10      return FALSE;
11  t_db ← GetDBContent(t_results);
12  o_db ← GetDBContent(o_results);
13  if t_db ≠ o_db then
14    ReportBug();
15    return FALSE;
16  return TRUE;
17 Function OracleGen(test_case, graph):
18  oracle_test_case ← [];
19  tmp_graph ← graph;
20  while HasNode(tmp_graph) = TRUE do
21    nodes ← GetZeroIndegreeNodes(tmp_graph);
22    node ← RandomlySelect(nodes);
23    stmt ← GetStmtFromNode(node, test_case, graph);
24    PushToList(oracle_test_case, stmt);
25    RemoveNode(tmp_graph, node);
26  return oracle_test_case;

```

test case, its corresponding acyclic SDG, and its transaction execution results (Step (7)). The graph decycling always converges, because it cannot indefinitely remove statements from the test case where the number of statements is finite.

**Oracle Checking.** Algorithm 1 shows the workflow of oracle checking. The inputs to this workflow are the refined *test case*, the acyclic *graph*, and the transaction execution results *t\_results* from graph decycling. We first generate a semantically-equivalent test case, that is, *oracle\_test\_case*, using topological sorting (Line 2). Specifically, we initialize *oracle\_test\_case* as an empty sequence and *tmp\_graph* as *graph* (Line 18–19). In each round, we randomly select one node whose in-degree—the number of edges directed into the node—is zero and append the statement corresponding to this node to the end of *oracle\_test\_case* (Line 21–24). Then, we remove the node from *tmp\_graph* and delete the edges incident to this node (Line 25). The loop terminates if all nodes in *tmp\_graph* are removed (Line 20), and the process returns the constructed *oracle\_test\_case* (Line 26). After *oracle\_test\_case* is available, we initialize the target DBMS and execute the statements in *oracle\_test\_case* in order without transactions (Line 3–4). According to Theorem 1, for any correct DBMS implementation, *test\_case* and *oracle\_test\_case* should produce the same result. We first compare each statement output (Line 5–10). If their outputs differ, we have found a bug. If these comparisons succeed, we further check whether the final database contents are the same (Line 11–15). If they are different, a bug is also found.

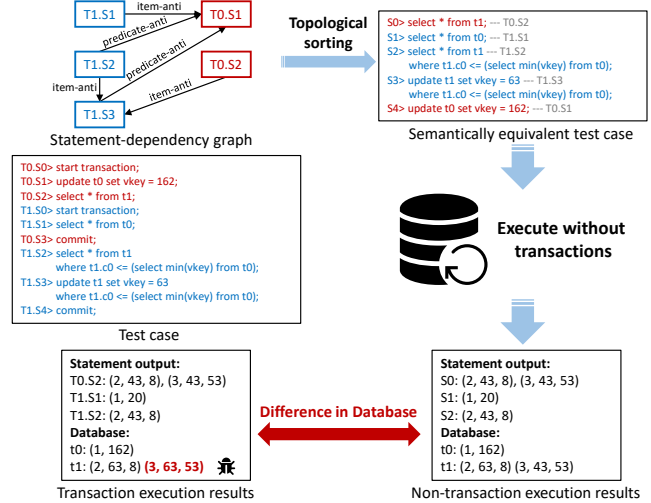


Figure 9: Oracle checking for the bug in Figure 1.

Figure 9 shows how we perform oracle checking on the test case in Figure 1(a). We first perform topological sorting on the acyclic statement-dependency graph. In the first round, T1.S1, T1.S2, and T0.S2 have zero in-degree, so we randomly pick one of them, for example, T0.S2. Then, T1.S1 and T1.S2 are picked. After T0.S2 and T1.S2 are removed from the graph, the in-degree of T1.S3 becomes zero, and T1.S3 is picked. Finally, T0.S1 is chosen as it is the only node in the graph. Therefore, the sorted statement sequence is [T0.S2, T1.S1, T1.S2, T1.S3, T0.S1]. Then, we execute the statement sequence without transactions and record the statement outputs and final database contents. These results are compared to the results produced by the original test case. We first check their statement outputs, which turn out to be the same. Then, we check their final database contents. Because their database contents are different on the value of one of the rows in table t1, we have found a bug.

## 4 Implementation

Based on our approach, we realized a tool, TxCheck, on top of SQLsmith [43]. The overall codebase consists of 14k lines of C++ code, where we implemented our approach with 3.5k lines (not including the code for supporting DBMSs).

Figure 10 shows the architecture of TxCheck. To test a DBMS, TxCheck first randomly generates a test case, which will be instrumented by SQL-level instrumentation. The instrumented test case is then refined by graph decycling to eliminate cycles in its statement-dependency graph, and by blocking scheduling to make sure that the instrumented statements will not be reordered by the blocking mechanism of the tested DBMS. TxCheck uses the refined test case and its transaction execution results to construct an oracle, which is a test case that is not wrapped as transactions (e.g., the test case shown in Figure 1(b)), but should produce the same results

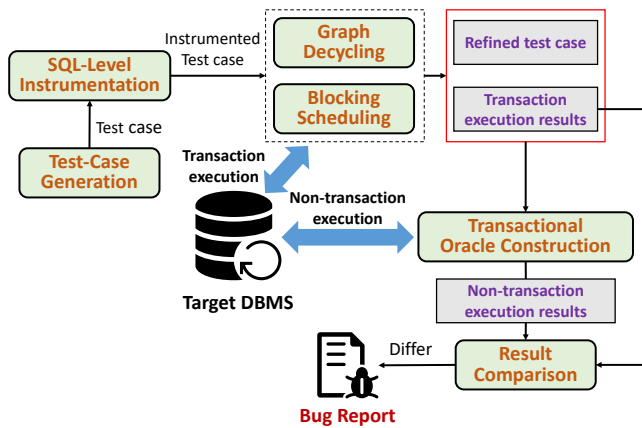


Figure 10: Architecture of TxCheck.

as the refined test case, according to Theorem 1. TxCheck then checks whether their results are indeed the same. If their results differ, TxCheck reports a bug. The following describes the implementation of TxCheck in detail.

**Test-case Generation.** TxCheck randomly generates a test case consisting of multiple transactions, and randomly determines the order in which the statements in these transactions are executed. For example, in Figure 1(a), T0 and T1 are the generated transactions, and [T0.S0, T0.S1, T0.S2, T1.S0, T1.S1, T0.S3, T1.S2, T1.S3, T1.S4] is the execution order for the statements in these transactions. The generated statements follow the constraints described in Section 3.2. As we do not restrict the statement format, TxCheck can apply other approaches to implementing statement generation. In this paper, we use SQLsmith [43] as the statement generator.

**Transaction Execution.** For each transaction in a test case, TxCheck sets up a client session that is responsible for issuing the statements of a transaction to which it is assigned. To avoid introducing non-determinism from concurrent executions, TxCheck sends statements to the DBMS following the order determined in the test-case generation. The order might be updated by block scheduling. After sending a statement, TxCheck sends the next statement only after the DBMS indicates that its execution is completed or blocked. While some concurrency bugs may be missed, sequential execution makes it significantly easier to reproduce bugs, which is generally appreciated by developers.

**Non-transaction Execution.** For each test case that is not wrapped as transactions (e.g., the test case in Figure 1(b)), TxCheck sets up only one client session for sequentially issuing the statements in the test case.

**Blocking Scheduling.** When statements in different transactions try to access the same data, a DBMS may block some of these statements to schedule transaction execution, which can disrupt the inserted statements of SQL-level instrumentation. Figure 11(a) shows an example in MySQL using *Repeatable Read* isolation level. T1.S1.BWR is the *BWR* statement of

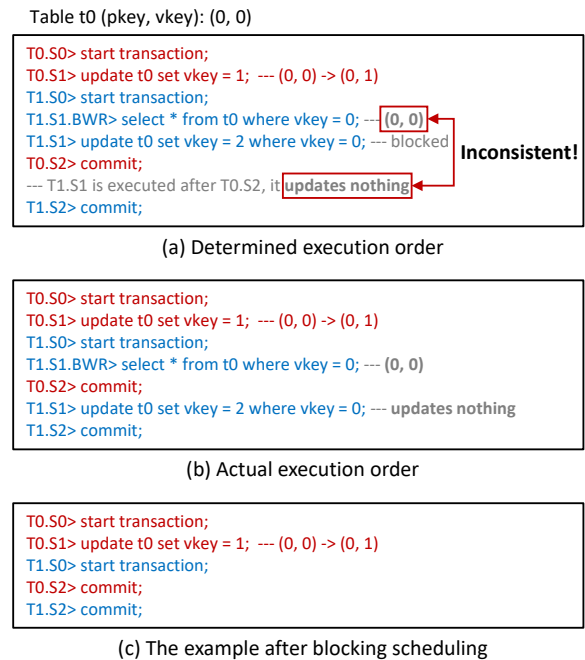


Figure 11: Example of MySQL blocking mechanism (in *Repeatable Read*) and blocking scheduling.

T1.S1. The DBMS executes T1.S1.BWR and outputs 1 row, and then tries to execute T1.S1. T1.S1 is blocked, because it tries to update the items that are being updated by T0.S1. The DBMS continues to execute T0.S2. After T0.S2 is executed, transaction T0 is finished, and then T1.S1 is executed automatically by the DBMS. T1.S1 updates nothing because there is no row whose vkey is 0. By design, T1.S1.BWR should output the items that will be updated by T1.S1. However, their results are inconsistent because T1.S1.BWR is executed before T0 commits but T1.S1 is executed after T0 commits. Figure 11(b) shows the actual execution order of the example. T1.S1.BWR and T1.S1 are separated by the COMMIT of T0.

To address the issues caused by the blocking mechanism of DBMSs, TxCheck adapts blocking scheduling, which makes sure that the inserted statements and the target statements will not be separated. TxCheck first executes statements according to the determined execution order. It records the actual statement execution order, which may be different from the determined order because some statements may be blocked. It checks whether there are situations where the inserted statements and their corresponding target statements are executed apart. It deletes the inserted statements and target statements in such situations. For example in Figure 11, T1.S1.BWR and T1.S1 are deleted. The refined test case is executed again following the recorded execution order in the last round. In the new execution, if all the target statements and corresponding inserted statements are executed adjacently, the blocking scheduling ends. Otherwise, it deletes statements according to the newly recorded real execution order and executes the re-



finer test case again. This process always converges, because it cannot delete statements indefinitely. Figure 11(c) shows the example refined by blocking scheduling.

**DBMS Support.** TxCheck can be easily adapted to test specific DBMSs. On average, we use 650 lines of code to support one DBMS. Each tested DBMS should provide interfaces to set up the DBMS, connect to the DBMS, shut down the DBMS, send statements in transaction sessions, and obtain execution results. In addition, if a DBMS can block statements in transaction execution, TxCheck needs to be provided with an interface for determining whether a statement is blocked. Setting a timeout for statements is an alternative way to check for blocking situations. However, it is inaccurate, because the DBMS may just be executing a long-running statement.

**Isolation Bug Detection.** TxCheck can also find isolation bugs, because statement dependencies can be easily converted to transaction dependencies according to their definitions. For example, if a statement  $S_i$  in transaction  $T_i$  depends on statement  $S_j$  in transaction  $T_j$ ,  $T_i$  depends on  $T_j$ . Therefore, TxCheck can convert statement-dependency graphs to transaction-dependency graphs. Then, TxCheck detects bugs that violate their isolation levels according to graph restrictions [1, 2, 5, 6]. However, because TxCheck may infer spurious predicate dependencies (Section 3.2), which introduce false alarms of isolation bugs, we only consider item dependency, which is accurate, in isolation bug detection.

**Memory Bug Detection.** As TxCheck involves both transaction and non-transaction executions, memory bugs triggered with or without transactions can be detected by TxCheck. We use ASan [40] as its memory bug checker.

## 5 Evaluation

We have evaluated TxCheck on three DBMSs, namely TiDB [46], MySQL [30], and MariaDB [29]. These DBMSs are widely used by industry and extensively tested by DBMS fuzzers [21, 22, 25, 35–37, 43, 53]. According to DB-Engines Ranking [13], MySQL is the second most popular relational DBMS, MariaDB the 8th, and TiDB the 49th. The GitHub repositories of TiDB, MySQL, and MariaDB have been starred more than 32K, 8K, and 4K times, respectively, demonstrating their popularity and maturity. We perform our evaluation on Ubuntu 20.04 with a 64-core AMD Epyc 7742 CPU at 2.25G Hz and 256GB RAM.

We evaluated TxCheck on the latest available releases of the targeted DBMSs. Specifically, for TiDB, we tested versions 5.4.0, 6.1.0, and 6.3.0, for MySQL, versions 8.0.28 and 8.0.30, and for MariaDB, versions 10.8.3 and 10.10.1. We tested MySQL and MariaDB under *Read Committed*, *Repeatable Read*, and *Serializability*, respectively. We did not test *Read Uncommitted* because it does not satisfy Assumption 3 (see Figure 7). TxCheck can be used to test *Read Committed*

Table 1: Numbers of bugs found by TxCheck and their status

DBMS	Found	Confirmed	Known	Fixed
TiDB	23	19	1	9
MySQL	18	18	1	3
MariaDB	15	15	4	1
<b>Total</b>	56	52	6	13

in MySQL (as also in MariaDB) because it supports consistent nonlocking reads [31] (e.g., read operations of SELECT), thus satisfying Assumption 3. We tested TiDB with its optimistic transaction mode and *Snapshot Isolation*, which is the only isolation level compatible with this mode [48]. We did not test the pessimistic transaction mode of TiDB [47], which does not satisfy Assumption 3.

We used TxCheck to continuously test the targeted DBMSs for three months; we stopped and restarted TxCheck only when we improved TxCheck with new SQL features. In general, TxCheck was able to find new bugs within several days after we implemented new features; however, certain bugs took more time to trigger (e.g., one or two weeks).

### 5.1 Bug Detection

As shown in Table 1, TxCheck found 56 unique bugs, including 23 in TiDB, 18 in MySQL, and 15 in MariaDB. Among them, 52 bugs were confirmed, 18 fixed, and 6 known.

**Bug Severity.** Regarding the 23 bugs found in TiDB, two were classified as *Critical* bugs, while 7 as *Major* bugs. The other bugs found in TiDB were assigned low severity (e.g., *Minor*). In MariaDB, all the found bugs were classified as *Critical* (9) or *Major* (6). Most of the bugs reported to the MySQL developers are confidential due to security concerns, so their severity is unavailable. 8 CVEs have been assigned to these security-related bugs. Additionally, 4 bugs posted publicly were classified as *Severe*. These results demonstrate that TxCheck is practical and effective in detecting critical bugs in production DBMSs.

**Bug Classification.** We classify the 56 bugs according to their root causes. Table 2 shows the results. The class "Transaction" includes the bugs found in transaction executions, and "Non-transaction" the bugs triggered in non-transaction executions, i.e., when the semantically-equivalent test cases are executed (see Section 3.3). The column "Crash" shows the number of bugs that crash DBMS servers, and "Oracle" refers to the bugs that are identified by oracle checking.

In total, TxCheck found 30 bugs triggered in transaction executions, which demonstrates TxCheck's capability for finding real transactional bugs in DBMSs. Among these bugs, 19 were identified by our oracle, and 11 crashed DBMS servers. In addition, 26 bugs were identified in the non-transaction executions, among which 23 crashed DBMS servers and 3

Table 2: Classifying the detected bugs

DBMS	Transaction		Non-transaction	
	Oracle	Crash	Oracle	Crash
TiDB	11	4	1	7
MySQL	4	2	0	12
MariaDB	4	5	2	4
<b>Total</b>	19	11	3	23

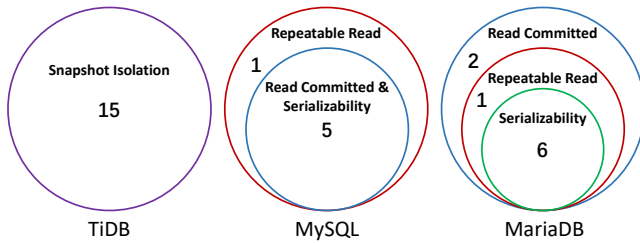


Figure 12: Venn diagrams showing the number of bugs found at different isolation levels. Bugs associated with a smaller circle of isolation level can also be found at larger circles of isolation levels.

were identified by oracle checking. Note that several bugs triggered in non-transaction executions make the execution results incorrect and different from the ones in transaction executions. This demonstrates that TxCheck can also detect incorrect behaviors triggered in non-transaction executions.

**Bugs at Different Isolation Levels.** For the 30 transactional bugs, Figure 12 shows the isolation levels where they are triggered. All 15 TiDB bugs are identified at *Snapshot Isolation*, which is the only tested isolation level in TiDB. In MySQL, 5 bugs can be found at all three tested isolation levels, while 1 only at *Repeatably Read*. In MariaDB, 6 bugs can be detected at all three tested isolation levels, while 1 at both *Read Committed* and *Repeatably Read*, and 2 only at *Read Committed*. Note that several transactional bugs are independent of isolation guarantees, which can be effectively detected by TxCheck at various levels.

## 5.2 Comparison with State of the Art

We demonstrate the advantages of our approach by (1) checking whether TxCheck can find new transactional bugs that cannot be found by the state of the art, and (2) discussing selected interesting bugs to show the effectiveness of TxCheck.

For comparison, we analyze the bug-triggering test cases of the 19 transactional bugs found by our oracle checking. We reduce each test case to a minimal bug-inducing version before analysis. Given that there exists no approach for finding general transactional bugs, we select ELLE [4] as competing tool (part of the prevalent testing framework Jepsen [18]). Elle is the state-of-the-art black-box checker for finding isolation

Table 3: Feature analysis of the 19 bug-triggering test cases

ID	DBMS	Features		ELLE
		Complex	Predicate	
1	TiDB	✓	✓	-
2	TiDB	✓	✓	-
3	TiDB	✓	✓	-
4	TiDB	-	-	✓
5	TiDB	-	-	-
6	TiDB	✓	✓	-
7	TiDB	✓	✓	-
8	TiDB	✓	✓	-
9	TiDB	✓	✓	-
10	TiDB	✓	✓	-
11	TiDB	✓	✓	-
12	MySQL	✓	✓	-
13	MySQL	✓	✓	-
14	MySQL	✓	✓	-
15	MySQL	✓	✓	-
16	MariaDB	✓	✓	-
17	MariaDB	✓	✓	-
18	MariaDB	✓	✓	-
19	MariaDB	✓	✓	-

bugs, which are a specific kind of transactional bugs.

As shown in Table 3, among the 19 test cases, 17 use both complex statements and predicates. ELLE cannot generate such test cases, because (1) the complex statements do not follow the test-case patterns of ELLE whose write operations can only append; and (2) ELLE does not support predicates. Figure 1 depicts one of such bugs triggered by complex statements and predicates. Regarding the two bugs that do not involve complex operations and predicates, ELLE can find only one of them. We also analyze 11 transactional bugs that crash DBMS servers and find that all of them involve complex statements and predicates, for which ELLE lacks support. In the following, we first illustrate the only bug that can be found by both TxCheck and ELLE. Then, we discuss three representative bugs that are missed by ELLE.

**TiDB Bug: Isolation Violation.** Figure 13 shows a bug-triggering test case. ELLE can only detect this bug from the 56 bugs found by TxCheck as the corresponding test case does not contain predicates or complex operations. This bug triggers a prohibited phenomenon, *G-Slb*, in Snapshot Isolation [1, 2] used by TiDB. *G-Slb* is an anomaly where the transaction-dependency graph contains a cycle with exactly one anti-dependency edge. To find this bug, TxCheck converts the constructed statement-dependency graph to a transaction-dependency graph (see Section 4) and checks whether there is any prohibited phenomenon. This bug-finding process illustrates that TxCheck can also find isolation bugs.

**MySQL Bug: Aborted Transactions Have Effects.** Figure 14 shows a test case with two interleaving transactions. Transaction  $T_1$  inserts four items into table  $t_0$  and then rollbacks. Transaction  $T_0$  updates the items that satisfy a complex

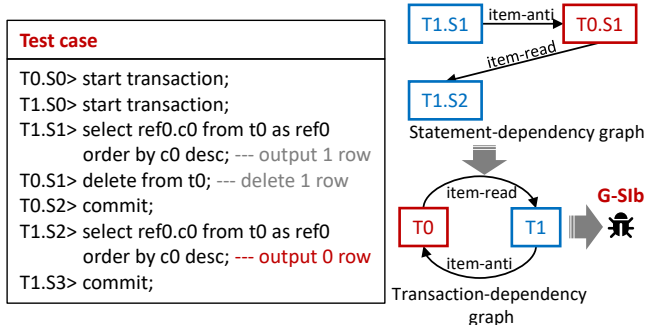


Figure 13: A test case violates *Snapshot* Isolation in TiDB.

Test case	Oracle
T0> start transaction; T1> start transaction; T1> insert into t0 values (141, 210000, ..., 74), ..., (141, 213000, ..., null); T1> rollback; T0> update t0 set vkey = 116 where t0.c5 not in ( select subq0.c0 as c0 from (select ...) as subq0 where subq0.c0 < (select ...) order by c0 asc); --- update 39 rows ✘ T0> commit;	update t0 set vkey = 116 where t0.c5 not in ( select subq0.c0 as c0 from (select ...) as subq0 where subq0.c0 < (select ...) order by c0 asc); --- update 0 row

Figure 14: An aborted transaction affects the results of a committed transaction in MySQL.

predicate (*i.e.*, the **WHERE** clause of the **UPDATE** statement) from table  $t_0$  and then commits. This test case is semantically-equivalent to executing the **UPDATE** statement on the same databases as the aborted transaction must not cause any visible side effects [1, 41]. However, MySQL produces different results: one test case updates 39 rows while the other zero rows. ELLE cannot find this bug as it involves predicates and complex operations, for which ELLE lacks support.

**TiDB Bug: Incorrect Transactional Calculation.** As shown in Figure 15, the test case contains only one transaction and uses only simple operations without predicates. ELLE can generate such a test case, at least conceptually. However, ELLE cannot find this bug, because it does not violate any isolation specification rather than makes the DBMS return incorrect results. TxCheck finds this bug by constructing semantically-equivalent test cases.

**MariaDB Bug: Crash Caused by Transactions.** As shown in Figure 16, the test case contains two interleaved transactions. Transaction T1 first inserts a couple of items into table  $t_0$ . Then, transaction T0 executes a **DELETE** statement with a complex **WHERE** clause as its predicate. The deletion is blocked because its predicate matching references certain items of table  $t_0$ , which have just been updated by the **INSERT** statement of T1. Only after T1 commits or aborts, can the **DELETE** statement be unblocked. While the deletion of T0 is blocked, transaction T1 executes a simple **UPDATE** statement, which eventually crashes the MariaDB server. This bug is

Test case	Oracle
T0> start transaction; T0> update t_0 set c_0 = t_2.c_1; T0> select count(c_2) from t_0; --- output 39 ✘ T0> commit;	update t_0 set c_0 = t_2.c_1; select count(c_2) from t_0; --- output 36

Figure 15: A test case makes TiDB return incorrect results.

Test case
T0> start transaction; T1> start transaction; T1> insert into t0 (vkey, pkey, c0) values (89,188000,40), ..., (97, 230000, 9); T0> delete from t1 where exists ( select ref0.c0 from t2 as ref0 where t1.c0 not in ( select ref3.vkey as c0 from (t0 as ref2 left outer join t2 as ref3 on (ref2.vkey = ref3.vkey)) where ref3.pkey >= ref2.vkey)); --- blocked T1> update t2 set vkey = 99; --- crash ✘

Figure 16: Two transactions crash the MariaDB server.

due to a concurrency issue where one of the threads performs complex operations that make the DBMS enter erroneous states. ELLE cannot find this bug as it does not support such complex operations involving predicates.

### 5.3 Design Choice Analysis

We had two considerations while designing our approach. First, can we use existing transaction-dependency graphs, *e.g.*, Directed Serialization Graph (DSG) [1, 2], instead of the proposed statement-dependency graphs? Second, when performing topological sorting, TxCheck randomly selects one node if there are multiple nodes with zero in-degree. Does the random strategy affect the results?

**Using Transaction-dependency Graphs.** We argue that using transaction-dependency graphs may miss bugs. A transaction-dependency graph in some isolation levels may have cycles as some transactions reference the items that other transactions have referenced. To construct transactional oracles, we must refine a test case to ensure the acyclicity of the associated graph. However, a test case may have an acyclic statement-dependency graph, but a cyclic transaction-dependency graph. Such test cases are unable to be topologically sorted at the transaction level, and thus interesting test cases may be discarded.

To demonstrate that using transaction-dependency graphs may miss bugs, we check the transaction-dependency graphs of the 19 transactional bugs found by our oracle checking. We first check the graphs built on minimized test cases and find that all the constructed transaction-dependency graphs miss cycles. It is unsurprising as, when minimizing the test cases, we delete all the unnecessary clauses and statements,

Table 4: Analysis of transaction-dependency graphs

DBMS	Test cases	Txn-cycle
TiDB	11	3
MySQL	4	1
MariaDB	4	2
<b>Total</b>	<b>19</b>	<b>6</b>

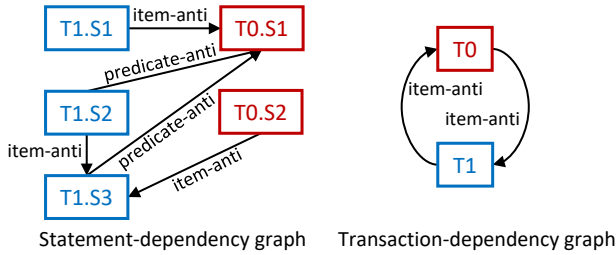


Figure 17: The statement-dependency and transaction-dependency graphs of the test case in Figure 1.

which makes the test cases reference much fewer items than the non-minimized ones. However, randomly generated test cases inevitably contain many redundant parts [28, 34]. To understand whether the test cases generated by TxCheck contain cycles in transaction-dependency graphs, we check the non-minimized test cases accordingly. Table 4 shows the results. The column *Txn-cycle* refers to the number of test cases that have cycles in transaction-dependency graphs.

The results show that 6 bug-triggering test cases have cycles in transaction-dependency graphs. If we use transaction-dependency graphs instead of statement-dependency graphs, these test cases are not suitable for constructing oracles as topological sorting cannot be performed for cyclic graphs. Hence, around one-third (6 out of 19) of the bugs would be missed. The test case in Figure 1 exemplifies such a bug. Figure 17 presents its corresponding statement-dependency and transaction-dependency graphs, respectively. The statement-dependency graph does not have cycles, so topological sorting can be performed at the statement level, which reveals the bug. Topological sorting is infeasible at the transaction level as the transaction-dependency graph is cyclic.

**Random Topological Sorting.** TxCheck topologically sorts statement-dependency graphs to construct oracles. If TxCheck encounters multiple nodes whose in-degrees are zero during sorting, it randomly selects one of them. In this way, TxCheck chooses only one of the topological sorting results to construct the oracle. For any correct DBMS, all the topological sorting results must be the same as the transaction execution results according to Theorem 1. However, when bugs are triggered, the test cases executed with transactions and some of the sorted test cases may produce the same, yet incorrect results. If TxCheck, unfortunately, chooses such sorted test cases, bugs may be overlooked.

Table 5: Analysis of topological sorting: \* labels the test cases that generate millions of topological sorting results, where we randomly select 10k to check whether they can trigger bugs

ID	DBMS	Sort	Trigger	ID	DBMS	Sort	Trigger
1	TiDB	1	1 (100%)	11	TiDB	1260	1224 (97%)
2	TiDB	32	32 (100%)	12	MySQL	6	6 (100%)
3	TiDB	12	12 (100%)	13	MySQL	1	1 (100%)
4	TiDB	-	-	14	MySQL	1	1 (100%)
5	TiDB	1	1 (100%)	15	MySQL	10k*	10k (100%)
6	TiDB	6	6 (100%)	16	MariaDB	2	2 (100%)
7	TiDB	6	6 (100%)	17	MariaDB	1	1 (100%)
8	TiDB	30	30 (100%)	18	MariaDB	1	1 (100%)
9	TiDB	96	96 (100%)	19	MariaDB	10k*	10k (100%)
10	TiDB	36	36 (100%)				

We show that such missed bugs are rare in practice. Typically, transactional bugs affect the results of transaction executions, while non-transaction executions of the topologically sorted test cases would not be affected. Therefore, most sorted test cases should execute correctly and can be used as oracles to reveal bugs. To demonstrate this, we analyze the 19 transactional bugs found by our oracle checking. The analyzed test cases are not minimized as we intend to obtain results that are close to those from the generated test cases in practice. Table 5 shows the results. The column *Sort* shows the number of all possible topological sorting results for the statement-dependency graphs. The column *Trigger* refers to the number of sorting results that successfully trigger the bugs.

Among the 19 test cases, the sorting results of 17 can stably trigger the bugs with 100% success rates. Bug 4 is found by checking transaction-dependency graphs, which do not involve topological sorting, as discussed in Section 5.2. The test case for bug 11 produces 1260 possible sorting results, among which 1224 can trigger the bug. It indicates that missed bugs can indeed happen with, however, a low probability (less than 3%). Two bug-triggering test cases (bugs 15 and 19) generate millions of sorting results. Both of them contain dozens of statements and involve few dependencies. When the dependency constraint is weak, the number of possible topological sorting will explode (*e.g.*, without any dependency, 12 statements can already amount to 12!, *i.e.*, over 480 million, sorting results). We randomly select 10K sorting results for each test case and find that all of them can trigger the bugs. These results show that randomly selecting one topological sorting result for oracle checking is practical and effective.

## 6 Discussion

**Test-case Generation.** TxCheck generates databases and transactions randomly for testing. Such random generation may be inefficient to explore corner test cases and thus may miss bugs. Fuzzing is a promising technique for generating infrequently executed test cases [3, 10, 16, 19, 27], and has been adopted in DBMS testing [21, 50, 53]. However, traditional fuzzing techniques cannot be directly utilized in DBMS



transaction testing. First, code coverage, which is commonly used as the fuzzing feedback, is not well suited because it cannot measure transaction interleavings. Second, random mutations used by most fuzzers are ineffective for generating transactions with complex data dependencies. One promising approach to addressing these challenges is to design new coverage feedback and mutation strategies following work on concurrency fuzzing [12, 20, 51].

**Predicate Handling.** It is challenging to recover transaction histories involving predicates [4, 44], for which we provide a possible solution. We instrument *Version-Set Read (VSR)* statements to capture items referenced by predicates by counting all items in the referenced tables. However, as discussed in Section 3.2, this method may overcount the referenced items because it is unnecessary that all items in the referenced tables would be referenced. Therefore, TxCheck may build spurious dependencies between statements. To mitigate this issue, one may utilize domain-specific knowledge. For example, we can customize *VSR* statements for each specific kind of statement used in transactions by referring to the corresponding SQL grammars and features.

**Data-intensive Transaction.** Existing work [21, 22, 35–37, 53] demonstrates that many DBMS bugs can be triggered without using much data. We follow this insight; each database generated by TxCheck generally contains 50-80 rows of data. However, some transactional bugs may hide in code only reached when intensive data is processed. To find such bugs, we plan to enable TxCheck to generate databases with large amounts of data. However, as TxCheck needs to reset database states after each test, its performance will be significantly degraded when TxCheck resets complicated databases. We plan to experiment with snapshot techniques to help improve testing performance by following and adapting existing work [24, 38].

## 7 Related Work

**Transaction Testing.** Transaction-testing approaches validate the correct uses of transactions in applications [14, 15] or the correct implementations of transaction support of DBMSs [4, 9, 11, 17, 44]. AGENDA [14, 15] tests DB-based applications that utilize transactions to perform certain tasks. It generates test cases according to user-provided specifications. A bug is reported if the application incorrectly constructs transactions that violate the provided specifications. The black-box isolation checkers ELLE [4], COBRA [44], and POLYSI [17] examine whether the transaction support of DBMSs functions correctly. ELLE generates transactions that use “append” operations as writes and can naturally recover their version order according to the list of values. COBRA and POLYSI focus respectively on validating the *Serializability* and *Snapshot Isolation* guarantees of transactions in DBMSs and develop several techniques (e.g., read-modify-

write transaction-based version order inferring, compact constraint encoding for SMT solving, and parallel hardware) to enable fast dependency inference. Unlike existing checkers, TxCheck focuses on testing the transaction support of DBMSs while relaxing the constraints on test-case patterns and enabling complex transaction generation. Moreover, TxCheck provides a practical solution to inferring predicate dependencies, a challenging problem in DBMS transaction testing.

**DBMS Testing.** Automated testing approaches have been proposed to find other types of bugs in DBMSs, such as logic bugs [35–37, 52], security bugs [21, 43, 50, 53], and performance bugs [22, 26]. SQLancer [42] is a well-known DBMS testing tool for detecting logic bugs, which integrates several approaches [35–37]. PQS [37] constructs queries that require DBMSs to return target items from manipulated databases: a logic bug is reported if the tested DBMS fails to return such items. TLP [36] designs some patterns to partition an original query into three separate queries, so that the union of the separated queries’ results must be the same as the original query’s result; otherwise, TLP reports a bug. Focusing on memory bugs, both SQUIRREL [53] and DynSQL [21] can generate more diverse test cases. SQUIRREL utilizes intermediate representations to model the structures of queries and the dependencies between statements. This enables SQUIRREL to generate queries containing multiple statements. By merging the query generation and query processing, DynSQL incrementally generates complex and valid queries using the state information of DBMSs. We also design TxCheck for tackling the oracle problem. However, TxCheck focuses on bugs triggered in transactional scenarios. In addition, with moderate test-case pattern constraints, TxCheck can handle complex test cases and expose deep transactional bugs.

## 8 Conclusion

We have presented a novel DBMS transaction testing approach, along with the practical tool TxCheck. Our approach is based on statement-level dependency graphs and can generate semantically-equivalent test cases to validate the transaction executions. TxCheck has found 56 unique bugs in three widely-used DBMSs, among which 52 have been confirmed and 18 fixed. Thanks to its generality and effectiveness, we expect TxCheck to help developers design and implement correct and reliable DBMS transaction support. Moreover, our approach could be utilized to infer predicate-related dependencies in recovering transaction histories.

## Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd, Tianyin Xu, for their valuable feedback on earlier versions of this paper. We also thank the DBMS developers for triaging and fixing our reported bugs.

## References

- [1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.
- [2] Atul Adya, Barbara Liskov, and Patrick E. O’Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.
- [3] American fuzzy lop. <https://github.com/google/AFL>.
- [4] Peter Alvaro and Kyle Kingsbury. Elle: Inferring isolation anomalies from experimental observations. In *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, pages 268–280, 2020.
- [5] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *Proceedings of the 39th International Conference on Very Large Databases (VLDB)*, pages 181–192, 2013.
- [6] Peter Bailis, Alan D. Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*, pages 27–38, 2014.
- [7] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. In *Proceedings of the 45th International Conference on Very Large Databases (VLDB)*, pages 2325–2338, 2019.
- [8] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.
- [9] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. In *Proceedings of the 2019 International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 165:1–165:28, 2019.
- [10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.
- [11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *Proceedings of the 26th International Conference on Concurrency Theory*, pages 58–71, 2015.
- [12] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium*, pages 2325–2342, 2020.
- [13] DB-Engines Ranking, Accessed in May, 2023. <https://db-engines.com/en/ranking>.
- [14] Yuetang Deng, Phyllis G. Frankl, and David Chays. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 78–87, 2005.
- [15] Yuetang Deng, Phyllis G. Frankl, and Zhongqiang Chen. Testing database transaction concurrency. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 184–195, 2003.
- [16] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, pages 2577–2594, 2020.
- [17] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, 2023.
- [18] Jepsen. <https://jepsen.io/>.
- [19] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, pages 2595–2612, 2020.
- [20] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.
- [21] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of the 32nd USENIX Security Symposium*.
- [22] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: automatic detection and diagnosis of performance regressions in database systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, pages 57–70, 2019.

- [23] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th International Conference on Very Large DataBases (VLDB)*, pages 96–107, 1994.
- [24] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security*, 17:2673–2687, 2022.
- [25] libFuzzer - a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>.
- [26] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Testing dbms performance with mutations. *arXiv preprint arXiv:2105.10016*, 2021.
- [27] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1949–1966, 2019.
- [28] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. In Robert Hirschfeld and Tobias Pape, editors, *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP)*, volume 166, pages 13:1–13:27, 2020.
- [29] MariaDB. <https://www.mariadb.org/>.
- [30] MySQL. <https://www.mysql.com/>.
- [31] MySQL Transaction Isolation Levels. <https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>.
- [32] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD)*, pages 677–689, 2015.
- [33] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the Association for Computing Machinery (JACM)*, 26(4):631–653, 1979.
- [34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 International Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.
- [35] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE)*, pages 1140–1152, 2020.
- [36] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. In *Proceedings of the 2020 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.
- [37] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.
- [38] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the 30th USENIX Security Symposium*, pages 2597–2614, 2021.
- [39] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.
- [40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, 2012.
- [41] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.
- [42] SQLancer. <https://github.com/sqlancer/sqlancer>.
- [43] SQLsmith: a random sql query generator. <https://github.com/ansel/sqlsmith>.
- [44] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 63–80, 2020.
- [45] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems*, 18(4):579–625, 1993.
- [46] TiDB. <https://www.pingcap.com/tidb/>.
- [47] Tidb pessimistic transaction mode. <https://docs.pingcap.com/tidb/stable/pessimistic-transaction>.
- [48] Tidb transaction isolation levels. <https://docs.pingcap.com/tidb/stable/transaction-isolation-levels>.

- [49] What is a Transaction? [http://msdn.microsoft.com/en-us/library/aa366402\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx).
- [50] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, pages 328–337, 2021.
- [51] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.
- [52] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *ISSTA '22*, pages 302–313. ACM, 2022.
- [53] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS)*, pages 955–970, 2020.



## A Definition for statement dependencies

We define statement dependencies in the same fashion as for the transaction dependencies defined by Adya *et al.* [1, 2].

### Definition 2 (Directly stmt-item-read-depends)

Statement  $S_j$  directly stmt-item-read-depends on statement  $S_i$  if  $S_i$  installs an item version  $x_i$  while  $S_j$  reads  $x_i$ .

### Definition 3 (Directly stmt-item-write-depends)

Statement  $S_j$  directly stmt-item-write-depends on statement  $S_i$  if  $S_i$  installs an item version  $x_i$  while  $S_j$  installs  $x$ 's next version (after  $x_i$ ) in the version order.

### Definition 4 (Directly stmt-item-anti-depends)

Statement  $S_j$  directly stmt-item-anti-depends on statement  $S_i$  if  $S_i$  reads an item version  $x_k$  while  $S_j$  installs  $x$ 's next version (after  $x_k$ ) in the version order.

### Definition 5 (Directly stmt-predicate-read-depends)

Statement  $S_j$  directly stmt-predicate-read-depends on statement  $S_i$  if  $S_j$  performs an operation  $r_j(P: \text{Vset}(P))$  while  $S_i$  installs an item version  $x_i$  that is included in  $\text{Vset}(P)$ .

### Definition 6 (Directly stmt-predicate-write-depends)

Statement  $S_j$  directly stmt-predicate-write-depends on statement  $S_i$  if either (1)  $S_j$  overwrites an operation  $w_i(P: \text{Vset}(P))$  performed by  $S_i$ , or (2)  $S_j$  executes an operation  $w_j(Q: \text{Vset}(Q))$  while  $S_i$  installs an item version  $x_i$  that is included in  $\text{Vset}(Q)$ .

### Definition 7 (Directly stmt-predicate-anti-depends)

Statement  $S_j$  directly stmt-predicate-anti-depends on statement  $S_i$  if  $S_j$  overwrites an operation  $r_i(P: \text{Vset}(P))$  performed by  $S_i$ .

### Definition 8 (Directly stmt-value-write-depends)

Statement  $S_j$  directly stmt-value-write-depends on statement  $S_i$  if either (1)  $S_i$  executes an operation  $w_i^{\text{value}}(E: \text{Vset}(E))$  where  $x_k$  is included while  $S_j$  install  $x$ 's next version (after  $x_k$ ) in version order, or (2)  $S_j$  executes an operation  $w_j^{\text{value}}(F: \text{Vset}(F))$  while  $S_i$  installs  $x_i$  that is included in  $\text{Vset}(F)$ . Here, statement  $S_i$  performs  $w_i^{\text{value}}(E: \text{Vset}(E))$  if  $S_i$  installs an item version whose values are based on expression  $E$  and the system (conceptually) reads all needed versions in  $\text{Vset}(E)$ .

## B Proof related to SQL-level instrumentation

Assumption 1 prohibits statements use old item versions while the newer ones are conceptually available. This can happen in distributed DBMSs when a new item version is produced but not well-synchronized, and thus the old version is still used in some machines. However, this work focuses on bugs in database engines deployed in local machines. Therefore, it is reasonable to assume that every item version is well-synchronized. Assumption 2 ensures that the inserted VSR

statements can correctly work. Assumption 3 ensures that the inserted BWR statements read the same item version used in the target statements.

**Lemma 1** Statement  $S_j$  directly stmt-item-read-depends on statement  $S_i \Leftrightarrow$  outputs of the AWR statement of  $S_i$  and outputs of  $S_j$  have intersections.

**Proof:** (1) Statement  $S_j$  directly stmt-item-read-depends on statement  $S_i \Rightarrow$  outputs of the AWR statement of  $S_i$  and outputs of  $S_j$  have intersections. Because  $S_j$  directly write-read depends on  $S_i$ , there is an item  $x$  such that  $S_i$  installs version  $x_i$  and  $S_j$  reads  $x_i$ .  $S_i$  installs version  $x_i$ , so  $x_i$  must be included in the output of AWR of  $S_i$ .  $S_j$  reads  $x_i$ , so  $x_i$  must be in the output of  $S_j$ . So the output of AWR of  $S_i$  and the output of  $S_j$  have intersections.

(2) Statement  $S_j$  directly stmt-item-read-depends on statement  $S_i \Leftarrow$  outputs of the AWR statement of  $S_i$  and outputs of  $S_j$  have intersections. Suppose  $x_k$  is one of the intersected item versions.  $x_k$  is in the output of AWR of  $S_i$ , so  $x_k$  must be installed by  $S_i$  because  $S_i$  is the only one statement that can assign the corresponding *VersionKey* value to  $x_k$  that matches the predicates of AWR of  $S_i$ .  $x_k$  is also in the output of  $S_j$ , so  $x_k$  is read by  $S_j$ . So  $S_i$  installs an item version  $x_k$  and  $S_j$  reads  $x_k$ . Therefore,  $S_i$  write-read depends on  $S_j$ .

Combining (1) and (2), we prove Lemma 1.

**Lemma 2** Statement  $S_j$  directly stmt-item-write-depends on statement  $S_i \Leftrightarrow$  outputs of the AWR statement of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections.

**Proof:** (1) Statement  $S_j$  directly stmt-item-write-depends on statement  $S_i \Rightarrow$  outputs of the AWR statement of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections. Because  $S_j$  directly write-write depends on  $S_i$ ,  $S_i$  installs a version  $x_i$  and  $S_j$  installs  $x$ 's next version (after  $x_i$ ) in the version order.  $S_i$  installs a version  $x_i$ , so  $x_i$  must be in the output of AWR of  $S_i$ . Suppose  $x_k$  is the version of item  $x$  that is used in the predicate matching of  $S_j$ .  $x_k$  must satisfy the predicate of  $S_j$  because  $S_j$  is going to install a new version for item  $x$ . As BWR of  $S_j$  uses the same predicate as  $S_j$ ,  $x_k$  must be in the output set of BWR of  $S_j$ . Suppose  $S_j$  is going to install  $x_j$ , there must be  $x_k \ll x_j$ . And  $x_j$  is  $x_i$ 's next version, so  $x_k$  is  $x_i$ , or  $x_k \ll x_i$ .

We assume that  $x_k \ll x_i$ . (a) If  $S_i$  and  $S_j$  are in the same transaction.  $x_i$  must be installed before  $x_j$ , which is installed by  $S_j$ , and  $x_k \ll x_i$ . Therefore, the BWR of  $S_j$  must read version  $x_i$  instead of  $x_k$ . Conflict. (b) If  $S_i$  and  $S_j$  are in different transactions,  $T_i$  and  $T_j$ . Because  $T_j$  reads  $x_k$ , and  $T_i$  installs  $x_i$ , which is after  $x_k$ ,  $T_i$  item-anti-depends on  $T_j$ . Because  $T_i$  installs  $x_i$ , and  $T_j$  installs  $x_j$  that are after  $x_i$ ,  $T_j$  item-write-depends on  $T_i$ . Conflict with Assumption 3.

Combining (a) (b), we get  $x_k \ll x_i$  in conflict. So  $x_k$  is  $x_i$ . So  $x_i$  is in the output of BWR of  $S_j$ . And  $x_i$  is in the output of

AWR of  $S_j$ . The output of AWR of  $S_i$  and the output of BWR of  $S_j$  have intersections.

(2) Statement  $S_j$  directly stmt-item-write-depends on statement  $S_i \Leftarrow$  outputs of the AWR statement of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections. Suppose  $x_k$  is one of the intersected item versions.  $x_k$  is in the output of AWR of  $S_i$ , so  $x_k$  must be installed by  $S_i$  because  $S_i$  is the only one statement that can assign the corresponding *VersionKey* value to  $x_k$  that matches the predicates of AWR of  $S_i$ .  $x_k$  is in the output of BWR of  $S_j$ , so  $x_k$  satisfies the predicate of BWR of  $S_j$ , and it satisfies the predicate of  $S_j$ . Therefore,  $S_j$  will install a new version of  $x$ . So  $S_i$  installs a version  $x_k$  and  $S_j$  installs  $x$ 's next version (after  $x_k$ ) in the version order, which means that  $S_j$  directly write-write depends on  $S_i$ .

Combining (1) and (2), we prove Lemma 2.

**Lemma 3** Statement  $S_j$  directly stmt-item-anti-depends on statement  $S_i \Leftrightarrow$  outputs of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections.

**Proof:** (1) Statement  $S_j$  directly stmt-item-anti-depends on statement  $S_i \Rightarrow$  outputs of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections. Because  $S_i$  directly stmt-item-anti depends on  $S_j$ ,  $S_i$  reads an item version  $x_k$  and  $S_j$  installs  $x$ 's next version (after  $x_k$ ) in the version order. Suppose  $x_h$  is the version of item  $x$  that is used in the predicate matching of  $S_j$ .  $x_h$  must satisfy the predicate of  $S_j$  because  $S_j$  is going to install a new version for item  $x$ . As BWR of  $S_j$  uses the same predicate as  $S_j$ ,  $x_h$  must be in the output of BWR of  $S_j$ . Suppose  $S_j$  is going to install  $x_j$ , there must be  $x_h \ll x_j$ . While  $x_j$  is  $x_k$ 's next version, so  $x_k$  is  $x_h$ , or  $x_h \ll x_k$ .

We assume that  $x_h \ll x_k$ : (a)  $S_i$  and  $S_j$  are in the same transaction.  $S_i$  must be before  $S_j$ .  $S_i$  reads version  $x_k$ , and thus  $S_j$  must use a version of  $x$  that is after  $x_k$  or equal to  $x_k$  (Assumption 1). However,  $S_j$  uses version  $x_h$ , and  $x_h \ll x_k$ . Conflict. (b)  $S_i$  and  $S_j$  are in different transactions,  $T_i$  and  $T_j$ . Suppose statement  $S_k$  of transaction  $T_k$  installs version  $x_k$ . If  $T_k$  is  $T_j$ ,  $S_k$  must be before  $S_j$  because  $S_j$  installs  $x_k$ 's next version. So  $x_k$  is visible to  $S_j$ , and  $S_j$  must use  $x_k$  instead of  $x_h$  ( $x_h \ll x_k$ ). Conflict. So  $T_k$  is not  $T_j$ . Because  $T_j$  reads  $x_h$ , and  $T_k$  installs  $x_k$  that are after  $x_h$ , so  $T_k$  item-anti-depends on  $T_j$ . And  $T_j$  installs  $x_j$  that are after  $x_k$ , so  $T_j$  item-write-depends on  $T_k$ . Conflict with Assumption 3.

Combining (a) (b), we get  $x_h \ll x_k$  in conflict. So  $x_k$  is  $x_h$ . So  $x_k$  is in the output of BWR of  $S_j$ . And  $x_k$  is in the output of  $S_i$ , The output of  $S_i$  and the output of BWR of  $S_j$  have intersections.

(2) Statement  $S_j$  directly stmt-item-anti-depends on statement  $S_i \Leftarrow$  outputs of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections. Suppose  $x_k$  is one of the intersected items.  $x_k$  is in the output of BWR of  $S_j$ , so  $x_k$  satisfies the predicate of BWR of  $S_j$ , and thus it satisfies the predicate of  $S_j$  ( $S_j$  and BWR of  $S_j$  use same version of  $x$ ). Therefore,  $S_j$  will install a version after  $x_k$ .  $x_k$  is in the output of  $S_i$ , so  $x_k$  is read by  $S_i$ . So  $S_i$  reads a item version  $x_k$  and  $S_j$  installs  $x$  with

a version after  $x_k$  in the version order. Therefore,  $S_i$  directly stmt-item-anti-depends on  $S_j$ .

Combining (1) and (2), we prove Lemma 3.

**Lemma 4** Statement  $S_j$  directly stmt-predicate-read-depends on statement  $S_i \Rightarrow$  outputs of the AWR statement of  $S_i$  and outputs of one of the VSR statements of  $S_j$  have intersections.

**Proof:** Statement  $S_j$  directly stmt-predicate-read-depends on statement  $S_i$ , which means that  $S_j$  performs an operation  $r_j(P: Vset(P))$ , and there is an item version  $x_i$  that is installed by  $S_i$  and  $x_i \in Vset(P)$ . Because  $x_i$  is installed by  $S_i$ , the  $x_i$  must be included in the output of AWR of  $S_i$ . Because VSRs of  $S_j$  outputs all item versions in the referenced tables, according to Assumption 2, all referenced item versions should be outputted by VSRs of  $S_j$ . Because  $x_i \in Vset(P)$ , at least one of the VSRs of  $S_j$  outputs  $x_i$ . So outputs of the AWR statement of  $S_i$  and outputs of one of the VSR statements of  $S_j$  have intersections. Proved.

**Lemma 5** Statement  $S_j$  directly stmt-predicate-write-depends on statement  $S_i \Rightarrow$  (1) outputs of one of the VSR statements of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections, or (2) outputs of the AWR statement of  $S_i$  and outputs of one of the VSR statements of  $S_j$  have intersections.

**Proof:** If statement  $S_j$  directly stmt-predicate-write-depends on statement  $S_i$ , according to Definition 6, it can be (1)  $S_j$  overwrites an operation  $w_i(P: Vset(P))$  performed by  $S_i$ , or (2)  $S_j$  executes an operation  $w_j(Q: Vset(Q))$  while  $S_i$  installs an item version  $x_i$  that is included in  $Vset(Q)$ .

For case (1),  $S_j$  overwrites operation  $w_i(P: Vset(P))$ , which means that  $S_i$  performs an operation  $w_i(P: Vset(P))$ , and there exists  $x_k \in Vset(P)$  that  $S_j$  installs  $x$ 's next version (after  $x_k$ ). Because  $x_k \in Vset(P)$ ,  $x_k$  must be in the output of one of the VSRs of  $S_i$ . Suppose  $x_h$  is the version of  $x$  that is used in  $S_j$  for predicate matching. Because  $S_j$  installs  $x$ 's next version (after  $x_k$ ),  $x_h$  must satisfy the predicate of  $S_j$  because  $S_j$  is going to install a new version for item  $x$ . As BWR of  $S_j$  uses the same predicate as  $S_j$ ,  $x_h$  must be in the output of BWR of  $S_j$  too. Suppose  $S_j$  is going to install  $x_j$ , there must be  $x_h \ll x_j$ . And  $x_j$  is  $x_k$ 's next version, so  $x_k$  is  $x_h$ , or  $x_h \ll x_k$ .

We assume that  $x_h \ll x_k$ : (a)  $S_i$  and  $S_j$  are in the same transaction.  $S_i$  must be before  $S_j$ .  $S_i$  uses version  $x_k$ , and thus  $S_j$  must use a version of  $x$  that is later than or equal to  $x_k$ . However,  $S_j$  uses  $x_h$  and  $x_h \ll x_k$ . Conflict. (b)  $S_i$  and  $S_j$  are in different transactions,  $T_i$  and  $T_j$ . Suppose statement  $S_k$  of transaction  $T_k$  installs version  $x_k$ . If  $T_k$  is  $T_j$ ,  $S_k$  must be before  $S_j$  because  $S_j$  install  $x_k$ 's next version. So  $x_k$  is visible to  $S_j$ , so  $S_j$  must use  $x_k$  instead of  $x_h$  as  $x_h \ll x_k$ . Conflict. So  $T_k$  is not  $T_j$ . Because  $T_j$  reads  $x_h$  (BWR of  $S_j$ ) while  $T_k$  installs  $x_k$  that are after  $x_h$ ,  $T_k$  item-anti-depends on  $T_j$ . Because  $T_j$  installs  $x_j$  that are after  $x_k$ ,  $T_j$  item-write-depends on  $T_k$ . Conflict with Assumption 3.

Combining (a) (b), we get  $x_h \ll x_k$  in conflict. So  $x_k$  is  $x_h$ . So  $x_k$  is in the output of BWR of  $S_j$ . And  $x_k$  is in the output of one of the VSRs of  $S_i$ , the outputs of one of the VSR statements of  $S_j$  and the output of BWR of  $S_j$  have intersections.

For case (2),  $S_j$  executes an operation  $w_j(Q: Vset(Q))$  while  $S_i$  installs an item version  $x_i$  that is included in  $Vset(Q)$ . Because  $x_i$  is installed by  $S_i$ , the  $x_i$  must be included in the output of AWR of  $S_i$ . Because  $x_i \in Vset(Q)$ ,  $x_i$  must be in the output of one of the VSRs of  $S_j$ . So the output of AWR of  $S_i$  and the output of one of the VSRs of  $S_j$  have intersections.

Combining (1) and (2), we prove Lemma 5.

**Lemma 6** Statement  $S_j$  directly stmt-predicate-anti-depends on statement  $S_i \Rightarrow$  outputs of one of the VSR statements of  $S_i$  and outputs of the BWR statement of  $S_j$  have intersections.

**Proof:** Statement  $S_j$  directly stmt-predicate-anti-depends on statement  $S_i$ , which means that  $S_j$  performs an operation  $r_i(P: Vset(P))$ , and there exists  $x_k \in Vset(P)$  that  $S_j$  installs  $x$ 's next version (after  $x_k$ ). Because  $x_k \in Vset(P)$ ,  $x_k$  must be in the output of one of the VSRs of  $S_i$ . Suppose  $x_h$  is the version of  $x$  that is used in  $S_j$  for predicate matching. Because  $S_j$  installs  $x$ 's next version (after  $x_k$ ),  $x_h$  must satisfy the predicate of  $S_j$  because  $S_j$  is going to install a new version for item  $x$ . As BWR of  $S_j$  uses the same predicate as  $S_j$ ,  $x_h$  must be in the output of BWR of  $S_j$  too. Suppose  $S_j$  is going to install  $x_j$ , there must be  $x_h \ll x_j$ . And  $x_j$  is  $x_k$ 's next version, so  $x_k$  is  $x_h$ , or  $x_h \ll x_k$ .

We assume that  $x_h \ll x_k$ : (a)  $S_i$  and  $S_j$  are in the same transaction.  $S_i$  must be before  $S_j$ .  $S_i$  uses version  $x_k$ , and thus  $S_j$  must use a version of  $x$  that is later than or equal to  $x_k$ . However,  $S_j$  uses  $x_h$  and  $x_h \ll x_k$ . Conflict. (b)  $S_i$  and  $S_j$  are in different transactions,  $T_i$  and  $T_j$ . Suppose statement  $S_k$  of transaction  $T_k$  installs version  $x_k$ . If  $T_k$  is  $T_j$ ,  $S_k$  must be before  $S_j$  because  $S_j$  install  $x_k$ 's next version. So  $x_k$  is visible to  $S_j$ , so  $S_j$  must use  $x_k$  instead of  $x_h$  as  $x_h \ll x_k$ . Conflict. So  $T_k$  is not  $T_j$ . Because  $T_j$  reads  $x_h$  (BWR of  $S_j$ ) while  $T_k$  installs  $x_k$  that are after  $x_h$ ,  $T_k$  item-anti-depends on  $T_j$ . Because  $T_j$  installs  $x_j$  that are after  $x_k$ ,  $T_j$  item-write-depends on Conflict with Assumption 3.

Combining (a) (b), we get  $x_h \ll x_k$  in conflict. So  $x_k$  is  $x_h$ . So  $x_k$  is in the output of BWR of  $S_j$ . And  $x_k$  is in the output of one of the VSRs of  $S_i$ , the outputs of one of the VSR statements of  $S_i$  and the output of BWR of  $S_j$  have intersections. Lemma 6 is proved.

**Lemma 7** Statement  $S_j$  directly stmt-value-write-depends on statement  $S_i \Rightarrow$  (1) outputs of one of the VSR statements of  $S_i$  and outputs of the BWR statement of  $S_j$  have overlapping parts, or (2) outputs of the AWR statement of  $S_i$  and outputs of one of the VSR statements of  $S_j$  have intersections.

**Proof:** If statement  $S_j$  directly stmt-value-write-depends on statement  $S_i$ , according to Definition 8, it can be (1)  $S_i$  executes an operation  $w_i^{value}(E: Vset(E))$  where  $x_k$  is included

while  $S_j$  installs  $x$ 's next version (after  $x_k$ ) in version order, or (2)  $S_j$  executes an operation  $w_j^{value}(F: Vset(F))$  while  $S_i$  installs  $x_i$  that is included in  $Vset(F)$ .

For case (1), because  $x_k \in Vset(E)$ ,  $x_k$  must be in the output of one of the VSRs of  $S_i$ . Suppose  $x_h$  is the version of  $x$  that is used in  $S_j$  for predicate matching. Because  $S_j$  installs  $x$ 's next version (after  $x_k$ ),  $x_h$  must satisfy the predicate of  $S_j$  because  $S_j$  is going to install a new version for item  $x$ . As BWR of  $S_j$  uses the same predicate as  $S_j$ ,  $x_h$  must be in the output of BWR of  $S_j$  too. Suppose  $S_j$  is going to install  $x_j$ , there must be  $x_h \ll x_j$ . And  $x_j$  is  $x_k$ 's next version, so  $x_k$  is  $x_h$ , or  $x_h \ll x_k$ .

We assume that  $x_h \ll x_k$ : (a)  $S_i$  and  $S_j$  are in the same transaction.  $S_i$  must be before  $S_j$ .  $S_i$  uses version  $x_k$ , and thus  $S_j$  must use a version of  $x$  that is later than or equal to  $x_k$ . However,  $S_j$  uses  $x_h$  and  $x_h \ll x_k$ . Conflict. (b)  $S_i$  and  $S_j$  are in different transactions,  $T_i$  and  $T_j$ . Suppose statement  $S_k$  of transaction  $T_k$  installs version  $x_k$ . If  $T_k$  is  $T_j$ ,  $S_k$  must be before  $S_j$  because  $S_j$  install  $x_k$ 's next version. So  $x_k$  is visible to  $S_j$ , so  $S_j$  must use  $x_k$  instead of  $x_h$  as  $x_h \ll x_k$ . Conflict. So  $T_k$  is not  $T_j$ . Because  $T_j$  reads  $x_h$  (BWR of  $S_j$ ) while  $T_k$  installs  $x_k$  that are after  $x_h$ ,  $T_k$  item-anti-depends on  $T_j$ . Because  $T_j$  installs  $x_j$  that are after  $x_k$ ,  $T_j$  item-write-depends on Conflict with Assumption 3.

Combining (a) (b), we get  $x_h \ll x_k$  in conflict. So  $x_k$  is  $x_h$ . So  $x_k$  is in the output of BWR of  $S_j$ . And  $x_k$  is in the output of one of the VSRs of  $S_i$ , the outputs of one of the VSR statements of  $S_i$  and the output of BWR of  $S_j$  have intersections.

For case (2), because  $x_i$  is installed by  $S_i$ , the  $x_i$  must be included in the output of AWR of  $S_i$ . Because  $x_i \in Vset(F)$ ,  $x_i$  must be in the output of one of the VSRs of  $S_j$ . So the output of AWR of  $S_i$  and the output of one of the VSRs of  $S_j$  have intersections.

Combining (1) and (2), we prove Lemma 7.

## C Proof for Theorem 1

**Inductive proof:**  $n$  is the number of statements in the statement-dependency graph (SDG).  $[T_{x1}S_{y1}, T_{x2}S_{y2}, \dots, T_{xn}S_{yn}]$  is the statement sequence executed within transactions.  $[S_{z1}, S_{z2}, \dots, S_{zn}]$  is the statement sequence generated by performing topological sorting on SDG.

when  $n = 1$ , obviously  $T_1S_1$  and  $S_1$  give the same results.

Suppose  $n = k$ , its SDG  $G_k$  is acyclic, and  $[T_{x1}S_{y1}, T_{x2}S_{y2}, \dots, T_{xk}S_{yk}]$  and  $[S_{z1}, S_{z2}, \dots, S_{zk}]$  produce the same results. When  $n = k + 1$ , we add a new statement at the end of the transactional statement sequence. Therefore, the sequence becomes  $[T_{x1}S_{y1}, T_{x2}S_{y2}, \dots, T_{xk}S_{yk}, T_{x(k+1)}S_{y(k+1)}]$ . We need to prove that Theorem 1 holds for  $k+1$  if the SDG is acyclic.

Because  $T_{x1}S_{y1}, T_{x2}S_{y2}, \dots$ , and  $T_{xk}S_{yk}$  are executed before  $T_{x(k+1)}S_{y(k+1)}$ , they are not affected by  $T_{x(k+1)}S_{y(k+1)}$ , and thus their execution results are the same as  $[T_{x1}S_{y1}, T_{x2}S_{y2}, \dots, T_{xk}S_{yk}]$  in the  $k$ -length case. So  $T_{x1}S_{y1}, T_{x2}S_{y2}, \dots$ , and  $T_{xk}S_{yk}$  still generate graph  $G_k$ . And then  $T_{x(k+1)}S_{y(k+1)}$

is executed and produces some dependencies to some of the executed statements. Therefore,  $G_{k+1}$  is a super graph of  $G_k$ . By performing topological sort, it generates several normal execution sequences. Suppose  $[S_{w1}, S_{w2}, \dots, S_{wk}, S_{w(k+1)}]$  is one of the sequences. Taking out the new statement  $S_{y(k+1)}$  from this sequence, we can get  $[S_{z1}, S_{z2}, \dots, S_{zk}]$ . Because we use topological sort, and  $G_{k+1}$  is a super graph of  $G_k$ ,  $[S_{z1}, S_{z2}, \dots, S_{zk}]$  follows the edges in  $G_{k+1}$  and thus follows the edges in  $G_k$ . Therefore,  $[S_{z1}, S_{z2}, \dots, S_{zk}]$  must be one of the topologically sorting results of  $k$ -length cases.

Now we consider the new statement  $S_{y(k+1)}$ . Suppose  $[S_{z1}, S_{z2}, \dots, S_{zp}, S_{y(k+1)}, S_{z(p+1)}, \dots, S_{zk}]$  is one of the topologically sorting results of  $k+1$ -length cases.  $[S_{z1}, S_{z2}, \dots, S_{zk}]$  is the topologically sorting results of  $k$ -length cases.

$S_{y(k+1)}$  will not affect the statements that are executed before it. Therefore, the  $S_{z1}, S_{z2}, \dots, S_{zp}$  produce the same results as they are in transaction execution (according to the  $k$ -length case). — **(Conclusion 1)**

Now, we need to prove: (1)  $S_{y(k+1)}$  produce the same results as  $T_{x(k+1)}S_{y(k+1)}$ ; and (2)  $S_{z(p+1)}, \dots, S_{zk}$  are not affected by  $S_{y(k+1)}$ , that is, they will produce the same results as they produce in transaction execution.

**(1)  $S_{y(k+1)}$  produce the same results as  $T_{x(k+1)}S_{y(k+1)}$ .**

**Proof:** If  $S_{y(k+1)}$  and  $T_{x(k+1)}S_{y(k+1)}$  produce different results, there must be at least one item  $x$  whose version  $x_i$  is referenced by  $S_{y(k+1)}$  and  $x_j$  is referenced by  $T_{x(k+1)}S_{y(k+1)}$ , and  $x_i$  is different from  $x_j$ . There are only two possible cases:

(a)  $x_i \ll x_j$ . Suppose  $T_{xj}S_{yj}$  installs item version  $x_j$ , so  $T_{x(k+1)}S_{y(k+1)}$  depends on  $T_{xj}S_{yj}$  because  $T_{x(k+1)}S_{y(k+1)}$  references the item version installed by  $T_{xj}S_{yj}$ . Therefore, topological sorting will put  $T_{xj}S_{yj}$  before  $T_{x(k+1)}S_{y(k+1)}$ . Suppose  $S_{zj}$  is the statement in sorted sequence corresponding to  $T_{xj}S_{yj}$ .  $S_{zj}$  is before  $S_{y(k+1)}$ , so  $S_{zj}$  should produce the same results as  $T_{xj}S_{yj}$  according to Conclusion 1. So  $S_{zj}$  also installs version  $x_j$ . So  $S_{y(k+1)}$  must reference the version of item  $x$  later than or equal to  $x_j$ . However,  $S_{y(k+1)}$  reference  $x_i$ , and  $x_i \ll x_j$ . Conflict.

(b)  $x_i \gg x_j$ . Suppose  $x_i$  is installed by  $S_{zi}$ . Because  $S_{y(k+1)}$  reference  $x_i$ ,  $S_{zi}$  must be before  $S_{y(k+1)}$ . According to Conclusion 1,  $S_{zi}$  produces the same results as it is in transaction execution. Suppose  $T_{xi}S_{yi}$  is the corresponding statement in the transaction execution.  $T_{xi}S_{yi}$  installs item version  $x_i$  while  $T_{x(k+1)}S_{y(k+1)}$  reference  $x_j$  that is older than  $x_i$ , so  $T_{xi}S_{yi}$  depends on  $T_{x(k+1)}S_{y(k+1)}$ . Therefore, topological sorting will put  $T_{xi}S_{yi}$  after  $T_{x(k+1)}S_{y(k+1)}$ , *i.e.*,  $S_{zi}$  is after  $S_{y(k+1)}$ , which is in conflict with that  $S_{zi}$  must be before  $S_{y(k+1)}$ .

Combining (a) and (b), we can get that there is no item that  $S_{y(k+1)}$  and  $T_{x(k+1)}S_{y(k+1)}$  reference its different version. Therefore,  $S_{y(k+1)}$  can produce only the same results as  $T_{x(k+1)}S_{y(k+1)}$ .

**(2)  $S_{z(p+1)}, \dots, S_{zk}$  are not affected by  $S_{y(k+1)}$ .**

**Proof:** We assume at least one of the statements in  $S_{z(p+1)}, \dots, S_{zk}$  is affected by  $S_{y(k+1)}$ . Suppose  $S_{zh}$  is the closest statement to  $S_{y(k+1)}$  among the statements that is affected by  $S_{y(k+1)}$ .

That is, there is not any statement between  $S_{zh}$  and  $S_{y(k+1)}$ , or statements between  $S_{zh}$  and  $S_{y(k+1)}$  should not be affected by  $S_{y(k+1)}$ . Because  $S_{zh}$  is affected, it must reference at least one item version that is installed by  $S_{y(k+1)}$ .

Suppose  $x_i$  is one of the item versions that are installed by  $S_{y(k+1)}$  and referenced by  $S_{zh}$ .  $S_{y(k+1)}$  and  $T_{x(k+1)}S_{y(k+1)}$  produce the same results, so  $T_{x(k+1)}S_{y(k+1)}$  also installs  $x_i$ . Suppose  $T_{xh}S_{yh}$  is the corresponding statement of  $S_{zh}$  in the transaction execution sequence. Because  $T_{xh}S_{yh}$  and  $S_{zh}$  are the same statement and thus use the same predicate,  $T_{xh}S_{yh}$  must reference one of the versions of item  $x$ . Suppose the item version is  $x_j$ .  $x_j$  must be different from  $x_i$  as  $T_{xh}S_{yh}$  is executed before  $T_{x(k+1)}S_{y(k+1)}$ , which is the last statement in transaction execution, and  $T_{xh}S_{yh}$  cannot reference an item version that has not been installed yet. There are only two possible cases:

(a)  $x_i \gg x_j$ .  $T_{xh}S_{yh}$  reference  $x_j$  while  $T_{x(k+1)}S_{y(k+1)}$  installs  $x_i$ , and  $x_i \gg x_j$ , so  $T_{x(k+1)}S_{y(k+1)}$  depends on  $T_{xh}S_{yh}$ . According to the topological sort,  $S_{zh}$  must be before  $S_{y(k+1)}$ . However,  $S_{zh}$  is after  $S_{y(k+1)}$ . Conflicts.

(b)  $x_i \ll x_j$ . Suppose  $x_j$  is installed by  $T_{xj}S_{yj}$ . Because  $T_{xj}S_{yj}$  installs  $x_j$  and  $T_{xh}S_{yh}$  reference  $x_j$ ,  $T_{xh}S_{yh}$  depends on  $T_{xj}S_{yj}$ .  $T_{xj}S_{yj}$  installs  $x_j$  while  $T_{x(k+1)}S_{y(k+1)}$  installs  $x_i$ , and  $x_i \ll x_j$ , so  $T_{xj}S_{yj}$  depends on  $T_{x(k+1)}S_{y(k+1)}$ . So  $T_{x(k+1)}S_{y(k+1)} \ll T_{xj}S_{yj} \ll T_{xh}S_{yh}$ . According to topological sorting,  $S_{y(k+1)}$  must be before  $S_{zj}$ , and  $S_{zj}$  must be before  $S_{zh}$ . Because  $S_{zh}$  is the closest statement that is affected by  $S_{y(k+1)}$ , and  $S_{zj}$  is before  $S_{zh}$ ,  $S_{zj}$  is not affected by  $S_{y(k+1)}$ . So  $S_{zj}$  will also install  $x_j$ . Therefore,  $S_{zh}$  should use a version later than or equal to  $x_j$ . However,  $S_{zh}$  references version  $x_i$  that is older than  $x_j$ . Conflicts.

Combining (a) and (b), we can get that there is no statement in  $[S_{z(p+1)}, \dots, S_{zk}]$  that is affected by  $S_{y(k+1)}$ . Therefore,  $S_{z(p+1)}, \dots, S_{zk}$  should produce the same results as they produce in transaction execution.

Combining (1) and (2), we can get that  $[T_{x1}S_{y1}, T_{x2}S_{y2}, \dots, T_{xk}S_{yk}, T_{x(k+1)}S_{y(k+1)}]$  and its topological sorting produce the same results. So for  $n = k + 1$ , the theorem still holds. Therefore, Theorem 1 is proved.





# Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate

Audrey Cheng<sup>†</sup>                      David Chu<sup>†</sup>                      Terrance Li<sup>†</sup>                      Jason Chan<sup>†</sup>  
Natacha Crooks<sup>†</sup>                      Joseph M. Hellerstein<sup>†</sup>                      Ion Stoica<sup>†</sup>                      Xiangyao Yu<sup>‡</sup>  
<sup>†</sup>UC Berkeley                      <sup>‡</sup>University of Wisconsin–Madison

## Abstract

Most caching policies focus on increasing object hit rate to improve overall system performance. However, these algorithms are insufficient for transactional workloads. In this work, we define a new metric, transactional hit rate, to capture when caching reduces latency for transactions. We present DeToX, a caching system that leverages transactional dependencies to make eviction and prefetching decisions. DeToX is able to significantly outperform single-object alternatives on real-world workloads and popular OLTP benchmarks, providing up to a 1.3x increase in transaction hit rate and 3.4x improvement in cache efficiency.

## 1 Introduction

To improve latency at scale, application developers often layer caching systems, such as Memcached [69] and Redis [2], over standard data stores. These systems traditionally optimize for *object hit rate*, or how often requested objects can be served from cache. Consequently, current caching policies fail to capture the transactional nature of many application workloads. On a production workload from Meta [26], we find that up to 90% of objects cached by least recently used (LRU) and least frequently used (LFU), two popular caching algorithms, do not have any impact on latency despite high object hit rates. Existing policies fail to capture the *all-or-nothing* property of transactions: all objects requested in parallel must be present in cache, or there will be little performance improvement because latency is dictated by the slowest access.

Accordingly, object hit rate is the wrong objective for transactional workloads. Instead, we propose a new metric, *transactional hit rate*, or how often objects requested in parallel can all be served from cache. This metric precisely captures when the cache reduces latency for transactions.

In this paper, we present DeToX, the first high-performance caching system that optimizes for transactional hit rate. In accordance with standard caching algorithms, DeToX assigns scores to objects and evicts those with the lowest values.

As such, its policy is easily adaptable to existing caching systems. To rank objects in the transactional context, DeToX leverages the following insight: objects accessed in parallel within the same transaction should be scored together since they must all be cached to reduce transactional latency.

While scoring keys together might seem simple, the structure of transactional workloads complicates matters. Unlike previous work on caching for parallel jobs [11] and web applications [7, 10, 18, 90, 91], transactions need to be modeled as non-trivial directed acyclic graphs (DAGs) of read and write operations [22, 94]. Crucially, some keys within a transaction are accessed in parallel, but others are not. Consequently, a transaction's latency is determined by its *critical length*, or the number of sequential accesses on its longest, non-cached path (transactional hit rate captures the reduction of critical length). Rather than considering all keys in a transaction together, we must focus on caching the *groups* of keys that reduce critical length.

Implementing a caching policy based on grouping presents several significant challenges. (1) For an arbitrary transaction, there can be an exponential number of groups, making scoring prohibitively expensive. (2) Identifying groups requires inferring transactional DAGs through static analysis, which may not always be possible. (3) Objects that are accessed by different transactions can belong to different groups, which have varying latency benefits if cached, and we need to capture these disparities.

We address each of these issues in DeToX. (1) To reduce the overhead of an exponential number of groups, we introduce the notion of *interchangeable* keys: if two keys can replace each other in any group and still reduce critical length, then they can be represented by the same group. Interchangeable keys drastically curb the number of groups that need to be scored. (2) When transactional DAGs are not accessible, we propose a simplified policy that dynamically infers groups based on which requests are executed in parallel (termed *levels*). (3) Finally, we account for group membership when scoring keys to ensure these values precisely reflect each object's contribution to transactional hits.

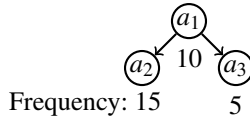


Figure 1: GetLinkedAccounts transaction.

Moreover, while our approach is primarily targeted at eviction, it also enables prefetching (Section 6). Our prefetching policy tracks dependencies within transactions to preemptively bring groups of items into the cache.

Our eviction and prefetching algorithms are implemented in DeToX, which presents a key-value API that supports drivers for Redis [2], Postgres [3], and TiKV [4]. We evaluate our system on real-world workloads from TAOBench [26], a social network benchmark that models Meta’s production workloads, as well as standard OLTP benchmarks (Epinions [37], SmallBank [87], and TPC-C [33]). Compared to single-object caching algorithms and systems, including ChronoCache [45], GDSF [27], LIFE [11], LFU, and LRU, our algorithm can achieve up to a 1.3x increase in *transactional hit rate*, leading to a 3.4x improvement in cache efficiency (defined as the least amount of cache space required to achieve a particular transactional hit rate). For a Redis-Postgres setup, this translates into 31% higher throughput and 30% lower latency.

Our transactional hit rate metric prioritizes *latency* and exposes a new trade-off in caching enabled by the cloud’s elastic resources: optimizing for latency versus reducing system load. In contrast, single-object policies focus on maximizing *object* hit rate to decrease load to the data store but do not always improve transaction request times.

In summary, we make the following contributions:

- We define a new metric, transactional hit rate, to evaluate the latency reduction of caching for transactions (Section 3).
- We provide the first formalization of transactional caching, and we prove that the problem is NP-Hard (Section 3.4).
- We present a new caching system, DeToX, that leverages transactional dependency information to optimize for transactional hit rate and significantly improve performance on popular workloads (Sections 4 – 8).

## 2 Motivation

In this section, we illustrate why single-object eviction algorithms perform poorly for transactional workloads. Specifically, we show that a well-known optimality result in caching does not hold for transactions and that popular caching algorithms achieve low transactional hit rates.

### 2.1 Object Hit Rate is Insufficient

Most existing cache eviction algorithms focus on maximizing object hit rate, or the fraction of single object requests served from cache. However, this approach fails to capture the

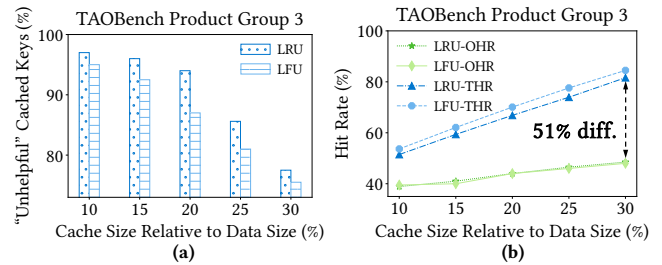


Figure 2: Single-object policy performance.

inter-object dependencies within transactions. Consider for example a simple transaction `GetLinkedAccounts` that returns secondary bank accounts  $a_2$  and  $a_3$  linked with a primary account  $a_1$  (Figure 1). This transaction must first read  $a_1$  before accessing both secondary accounts  $a_2$  and  $a_3$  in parallel. Thus,  $a_1$ ,  $a_2$  and  $a_3$  are all on the longest path of the transaction. If we cache  $a_1$ , we can reduce the end-to-end latency of the transaction. However, if we additionally cache  $a_2$ , the overall latency does not improve because we still need to access  $a_3$  from disk. In fact, caching either  $a_2$  or  $a_3$  individually does not improve performance; transaction latency remains equal to the case in which *neither* key was cached. On the other hand, caching *both*  $a_2$  and  $a_3$  does improve latency.

Transactions have an implicit *all-or-nothing* property on groups of objects that traditional caching algorithms fail to capture. This can lead popular eviction algorithms, such as LRU and LFU, to make poor caching decisions. Consider a situation in which, over all transactions,  $a_2$  is more frequently accessed than  $a_1$  and  $a_3$ . LFU and LRU would choose to evict  $a_1$  and  $a_3$  over  $a_2$ , resulting in no latency improvement for this transaction. In this case, a “hot” (frequently accessed) key  $a_2$  is requested in parallel alongside a “cold” (rarely accessed) key  $a_3$ . If all accesses of  $a_2$  are sent in parallel with requests to different cold keys, there is no benefit to caching  $a_2$  unless all these cold keys are cached. In effect, cold keys can “contaminate” (degrade the cacheability of) hot keys like  $a_2$ .

**Real-world workloads.** This observation is not limited to our simple example: we find that single-object eviction algorithms also perform poorly for complex, real-world workloads. Figure 2a illustrates that over 90% of cached keys do not have any impact on latency (“unhelpful” keys) for the Product Group 3 workload of TAOBench [26]. The root cause is simple: these algorithms optimize for *object* hit rate (OHR) rather than transactional hit rate (THR). As we see in Figure 2b, LRU and LFU achieve high object hit rates but up to 51% lower transactional hit rates. Transactions in this workload access either a combination of hot keys and warm keys, or hot keys and cold keys. Single-object algorithms, which use only individual object features to score keys, retain only hot keys but evict most warm keys and all cold keys. As a result, they achieve few transactional hits. A transactionally-aware policy would instead recognize that cold keys contaminate their associated hot keys and prioritize retaining only the hot and warm keys that are accessed together.

T	Keys accessed	Cache state	Optimal cache state
1	$a_1, a_2, a_3$	-	-
2	$a_4, a_5, a_6$	$a_1, a_2, a_3$	$a_1, a_2, a_3$
3	$a_4, a_5, a_7$	$a_1, a_4, a_5$	$a_1, a_2, a_3$
4	$a_1, a_2, a_3$	$a_1, a_4, a_5$	$a_1, a_2, a_3$

Figure 3: Non-optimality of Belady.

## 2.2 Optimality

Our observations also have theoretical implications. We find that Belady [16], the offline, optimal eviction algorithm for uniformly-sized objects does not make the best decisions for maximizing transactional hit rate. This policy evicts keys that are accessed furthest in the future but fails to take into account whether these keys generate transactional hits.

We prove that Belady is *not* optimal even for the simplest case of uniformly-sized transactions with uniformly-sized objects (Figure 3). In this example, we have four transactions with a cache size of 3.  $T_1$  and  $T_4$  access keys  $a_1, a_2, a_3$ , while  $T_2$  accesses  $a_4, a_5, a_6$  and  $T_3$  accesses  $a_4, a_5, a_7$ . Belady chooses to first cache  $a_1, a_2, a_3$  and then replaces the last two keys with  $a_4, a_5$  since these keys give object hits (but no latency reduction) for  $T_3$ . However, keeping  $a_2, a_3$  in the cache would lead to a transactional hit (and latency improvement) for  $T_4$ .

## 2.3 Towards a new approach

Our results highlight how single-object caching strategies yield low transactional hit rates by storing many unhelpful objects. Web caching algorithms suggest a way forward: they acknowledge the need to cache multiple objects together (e.g., page-level hit ratio) but only consider flat dependencies [11, 91]. In contrast, transactions can have complicated topologies with multiple levels of dependencies.

To develop a transactionally-aware caching system, we must address three challenges: (1) formalizing caching in the transactional context, including optimality analysis (Section 3), (2) identifying which groups of objects lead to transaction hits, given the potentially complex structure of transactions (Section 4.1), and (3) scoring the individual objects in these groups to determine which objects to store in the cache (Section 4.2). In our design, we are careful to emphasize compatibility with existing caching systems, such as Memcached and Redis, so that our approach can be easily implemented for greater applicability.

## 3 Transactional Caching

In this section, we formalize the transactional caching problem. We define a new metric, *transactional hit rate*, to capture the latency reduction of caching transactions.

```

1 id = SELECT cId FROM ACCOUNTS WHERE name = cName
2 s = SELECT savings FROM SAVINGS WHERE cId = id
3 c = SELECT checking FROM CHECKING WHERE cId = id
4 return s + c

```

Listing 1: Code for Figure 4. The dependencies for Lines 2 and 3 on the output of Line 1 are highlighted in red.

## 3.1 Transactions

Transactions consist of read and write requests that must be applied atomically [22]. Some of these operations are independent and can execute in parallel, while others are *dependent* on the result of preceding operations. For instance, a read operation may query a key determined by the return value of a previous operation. As a result, these operations must be run sequentially. In effect, transaction execution can be captured by a DAG of operations. More formally, we apply the notion of a logical dependency, generalizing the model from Wu et al. [94]:

**Definition 1** (*Logical dependency*). Given two operations  $tp$  and  $t$  of a transaction, an operation  $t$  is logically dependent on operation  $p$  if  $p$  determines the key or value accessed by  $t$ .

Traditionally, these dependencies are not captured by the system, which observes only sequences of reads and writes. In practice, these relationships can be captured statically through program analysis or specified at run time by the developer. Together, operations and logical dependencies define a transaction execution graph:

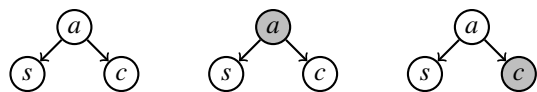
**Definition 2** (*Transaction execution graph*). For transaction  $T$ , a transaction execution graph  $G = (V, E)$  is a DAG, where each vertex in  $V$  represents a pair  $(x, X)$  of a read or write operation to key  $x$  in table  $X$ , and each edge in  $E$  represents a logical dependency between operations.

Each transaction execution graph corresponds to a transaction type:

**Definition 3** (*Transaction type*). Transactions of the same type have identical execution graphs when only considering tables.

We infer transaction execution graphs and their resulting types through static analysis, as done in prior work [36, 94]. Note that we only extract table accesses and graph structure; the individual keys accessed by transactions are known only at run time. As such, we make no assumptions about the DAG structure and support general-purpose, interactive transactions. For example, the SmallBank workload [87] contains the transaction types: Amalgamate, Balance, DepositChecking, SendPayment, TransactSavings, and WriteCheck. For the Balance transaction (Listing 1), requests to both the Savings ( $S$ ) and Checkings ( $C$ ) tables are dependent on the result of the read to the Accounts table ( $A$ ). The corresponding execution graph consists of three nodes, one for each operation, and logical dependencies  $r[A] \rightarrow r[S]$  and  $r[A] \rightarrow r[C]$ . While the reads to  $S$  and  $C$  are independent





(a) Cache state:  $\{\}$  (b) Cache state:  $\{a\}$  (c) Cache state:  $\{c\}$

Figure 4: SmallBank Balance transactions.

and can be executed in parallel, they cannot proceed until after the read to  $A$  finishes. At run time, a Balance transaction that reads the keys  $a, s, c$  from the tables  $A, S, C$  respectively can be mapped onto the same execution graph (Figure 4a).

## 3.2 Cache

The previous section presents the notion of a transaction, including the logical dependencies that constrain its execution. We now formalize how *caching* affects transactions, drawing from Abrams et al. [48] for notation.

**Definition 4 (Cache state).** A cache state is a set of keys  $C$  for which  $|C| \leq n$ , where  $n$  is the capacity of the cache.

In line with prior work [29], we assume that the cache state does not change for the duration of each transaction.

By assumption, objects are served with lower latency from the cache than from the underlying data store. We make the simplifying assumption that requests served from the cache have *zero* latency for notational simplicity (we explore the effects of varying cache latency in Section 8.6). Under this model, transaction latency is defined by the number of sequential, *non-cached* accesses. This corresponds to the longest path in the transaction’s execution graph  $G$ , excluding vertices with cached keys.

We formalize this notion as the critical length:

**Definition 5 (Critical length).** Given a transaction  $T$  with transaction execution graph  $G$ ,  $K$  number of keys, and cache state  $C$ , the critical length is the length of the longest path from any source vertex (no incoming edges) to any sink vertex (no outgoing edges), excluding vertices corresponding to keys in  $C$ . We define the function  $L: G \times 2^K \rightarrow \mathbb{N}$  for which  $2^K$  is the powerset of all keys, such that  $L(G, C)$  is the critical length.

Given a transaction  $T$  with execution graph  $G$ ,  $L(G, \{\})$  represents the length of the longest path in  $G$  when the cache is empty. For example, Figure 4a has longest paths  $\{r[a], r[c]\}$  and  $\{r[a], r[s]\}$  with critical length  $L(G, \{\}) = 2$ . Caching key  $a$  (Figure 4b) would shorten the critical length to  $L(G, \{a\}) = 1$ , as the longest paths are reduced to  $\{r[c]\}$  and  $\{r[s]\}$ . However, caching key  $c$  (Figure 4c) does not change the critical length, since  $\{r[a], r[s]\}$  remains the longest path with  $L(G, \{c\}) = 2$ . Informally, we refer to each length reduction as a *transactional hit*.

## 3.3 Transactional Hit Rate (THR)

Having defined the necessary formalisms for transaction latency and caching, we can now introduce transactional hit rate. Informally, this metric captures how much latency improves

when caching for transactions, much like how its single-object counterpart, object hit rate, does so for individual requests.

We first present THR in the context of a single transaction:

**Definition 6 (Individual transactional hit rate).** Given transaction  $T$  with execution graph  $G$  and cache state  $C$ , the individual transactional hit rate is  $\frac{L(G, \{\}) - L(G, C)}{L(G, \{\})}$ .

The difference in critical length represents the reduction in sequential, non-cached accesses after caching. We normalize this difference by dividing by the total critical length. This metric captures the impact of caching for the execution of a single transaction (note that if the transaction execution graph is a sequential list of dependent reads, then transactional hit rate is equivalent to object hit rate). We easily extend this definition to a sequence of transactions:

**Definition 7 (Transactional hit rate).** Given a sequence of transactions  $T_1, T_2, \dots, T_m$  with execution graphs  $G_1, G_2, \dots, G_m$  and the respective cache states at the time of execution  $C_1, C_2, \dots, C_m$ , the transactional hit rate is  $\frac{\sum_{i=1}^m (L(G_i, \{\}) - L(G_i, C_i))}{\sum_{i=1}^m L(G_i, \{\})}$ .

## 3.4 Optimality Analysis

Single-object caching is a well-studied problem and is known to be NP-Hard in the general case [29]. We show that the optimal transactional caching is NP-Hard through a reduction from variable-sized caching of single objects (proof in Appendix A). In summary, we reduce each variable-sized object of size  $X$  to a transaction with  $X$  unit-sized operations.

## 4 Group Identification and Scoring

Designing an optimal caching policy is impractical for transactional caching, since it would run in exponential time. Unfortunately, traditional heuristics perform poorly for transaction hit rate (Section 2) because they fail to identify the keys that must be cached as a *group* in order to yield a transactional hit. This notion of grouping is central to developing a transactionally-aware caching policy. We proceed in two steps: first, we identify which groups of keys lead to transactional hits when cached together (*group identification*). Next, we determine what scores should be assigned to each key within a group (*group scoring*).

Figure 5 gives an overview of DeToX. Our system first extracts transaction execution graphs (Section 3) from application code and identifies groups of table accesses (Section 4.1) at compile time. The number of groups that DeToX needs to consider can be reduced at compile time through the notion of *interchangeability* (Section 5.1). DeToX then scores groups based on key accesses at run time (Section 4.2). If application code is not available, DeToX constructs approximate groups at run time by using *levels* (Section 5.2).

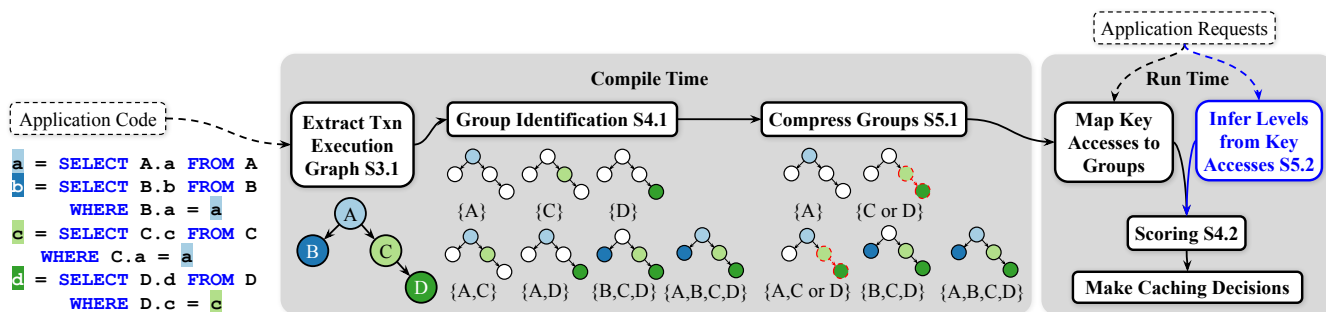


Figure 5: Overview of DeToX (gray boxes). Blue edges on the right represent the path taken if application code is not available.

## 4.1 Group Identification

Intuitively, a group is a set of keys that reduces critical length if cached together. Specifically, we define a *complete group* as one from which we cannot remove any key without increasing critical length. Completeness optimizes cache efficiency by storing the minimal subset of keys necessary to reduce latency. Formally:

**Definition 8 (Complete group).** Given a transaction  $T$  and its execution graph  $G$ , a complete group is a subset of keys  $g$  accessed in  $T$  such that  $\forall g' \subset g, L(G,g) < L(G,g')$ .

We identify complete groups of table accesses at compile time, using the transaction execution graphs  $G$  extracted via static analysis, as seen in Figure 5. A simple algorithm to identify groups is to iterate through the powerset of possible table accesses and compute their resulting reductions in critical length. These table accesses are replaced at run time with key accesses. The application passes along metadata with requests to indicate the corresponding vertex in the transaction execution graph of each key access.

Consider Figure 6a, which has a critical length of three (serial accesses of  $a, c, d$ ) and seven complete groups ( $\{a\}$ ,  $\{c\}$ ,  $\{d\}$ ,  $\{a, c\}$ ,  $\{a, d\}$ ,  $\{b, c, d\}$ ,  $\{a, b, c, d\}$ ). Note that  $\{c, d\}$  is not a complete group. If  $c$  and  $d$  are both cached, then the critical length is two (serial accesses of  $a, b$ ). However, only caching  $c$  already yields the same critical length (accesses to  $a, b, d$ ). Similarly,  $\{a, b\}$  is not a complete group, because it yields a critical length of two (serial accesses  $c, d$ ), which could also be achieved by just caching  $a$ .

In the worst case, the number of complete groups can be exponential in the size of the transaction, even for simple transaction topologies. Fortunately, many of these groups are, in fact, *equivalent*. We describe this notion more precisely in Section 5.1 and present an optimization that drastically reduces the number of groups that need to be considered.

## 4.2 Scoring

Caching policies typically assign scores to keys and evict keys with lower values. We adopt the same strategy by mapping complete groups to individual key scores at run time, as seen in Figure 5. This approach has two benefits: (1) we can draw from prior work on single-object caching

Parameter	Description
$\text{SCORE\_G}(\text{group})$	Score of a group
$F_{\text{group}}$	Set of all key frequencies in a group
$L_{\text{group}}$	Transactional hits of a group
$S_{\text{group}}$	Sum of key sizes in a group
$\text{SCORE\_K}(\text{key})$	Score of a key
$TS_{\text{key}}$	Sum of instance scores for a key
$F_{\text{key}}$	Frequency of a key
$A_{\text{global}}$	Global aging factor

Table 1: Scoring parameters.

algorithms, and (2) we minimize implementation changes needed for real-world caching systems.

### 4.2.1 Scoring a Group in a Single Transaction

We begin by assigning numerical scores to each group (*group scores*) with higher values representing groups that are more beneficial to cache. We draw inspiration from GDSF, a high-performing web caching algorithm [27]. GDSF considers three metrics to score keys: frequency (access count), recency, and size. Specifically, GDSF uses the following formula:  $\text{SCORE}_{\text{GDSF}}(\text{key}) = F_{\text{key}}/S_{\text{key}} + A_{\text{global}}$ , where  $F_{\text{key}}$  is frequency of the key,  $S_{\text{key}}$  is size of the key, and  $A$  is a global recency factor (described in Section 4.2.3). GDSF gives equal weight to each of these factors, and we follow this approach. We leverage frequency and size to score each group as follows (and incorporate recency into key scores in Section 4.2.3):

$$\text{SCORE\_G}(\text{group}) = \frac{\min(F_{\text{group}}) \times L_{\text{group}}}{S_{\text{group}}}$$

$F_{\text{group}}$  is a list of all key frequencies in the group.  $L_{\text{group}}$  is the number of transactional hits generated if this group is cached.  $S_{\text{group}}$  is the sum of all key sizes in the group. All scoring parameters can be found in Table 1. For the transactions in Figure 6 (which will be used as running examples), the group scores of each complete group for these transactions are shown in Figures 6b and 6d. The transaction in Figure 6a has keys  $a, b, c, d$  with frequencies of 1, 29, 99, and 50, respectively and sizes of 1. The score of group  $\{a, b, c, d\}$  is thus  $\frac{\min(1, 29, 99, 50) \times 3}{4} = 0.75$ .

**Frequency ( $F_{\text{group}}$ ).** Keys within a complete group may vary in frequency but must all be cached to yield a transactional

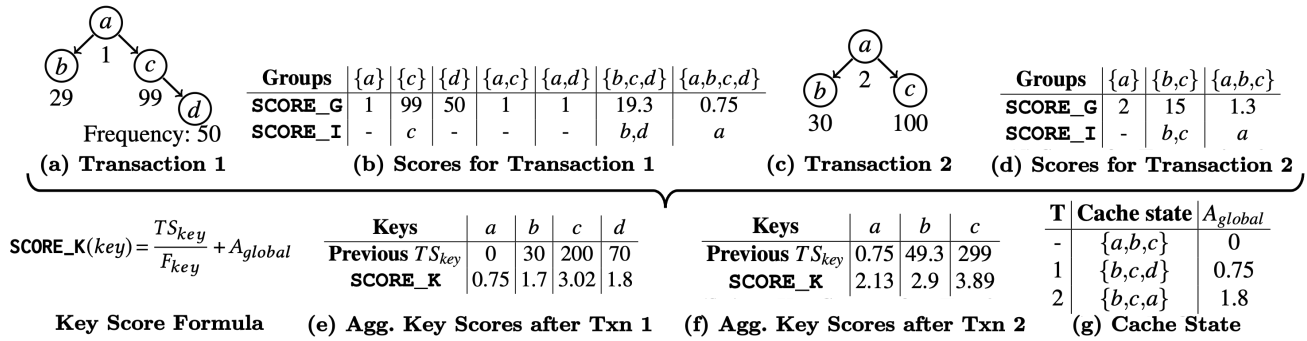


Figure 6: Example transactions and scores. Key sizes are 1, cache size is 3, and  $A_{global}$  starts at 0. The cache initially stores  $\{a,b,c\}$ .  $a$  is evicted after T1, and  $d$  is evicted after T2, with  $A_{global}$  updated on each eviction.

hit. For example, if a high-frequency key  $x$  is only associated with a group of keys  $\{y_1, \dots, y_k\}$  (each with much lower frequency than  $x$ ), then it is not beneficial to cache  $x$ . Essentially, the key with the minimum frequency determines the cacheability of the entire group. Thus, we take the minimum of all key frequencies in calculating the group score. Consider for instance the transaction in Figure 6c: key  $b$  is less frequently accessed than key  $c$  and drives down the frequency of the group  $\{b,c\}$  to  $\min(F_{group}) = \min(30, 100) = 30$ . In this example,  $b$  contaminates  $c$ .

**Critical length reduction ( $L_{group}$ ).** This parameter captures the reduction in critical length when caching a group ( $L_{group} = L(G, \{\}) - L(G, group)$ ). Other factors being equal, groups with greater reductions are better choices to cache and should thus be assigned a higher score.

**Size ( $S_{group}$ ).**  $S_{group}$  represents the cache space needed to store the group. Since all keys in a group must be present in cache to generate a transactional hit, THR is maximized by retaining groups of smaller sizes (more groups can be cached).

Next, we describe how to go from group scores to key scores.

#### 4.2.2 Scoring Across Groups in a Single Transaction

Mapping group scores to keys is challenging: for a given transaction, a key can belong to multiple complete groups, each with a separate group score (SCORE\_G). In this section, we focus on assigning scores to keys within a single transaction; we assign each key an instance score (SCORE\_I) based on one of its group scores. We combine instance scores across transactions in Section 4.2.3.

Our algorithm leverages the insight that out of all the keys in a transaction, the highest-scoring complete group is the most beneficial set of keys to cache. Thus, our protocol first finds the complete group with the highest group score SCORE\_G and sets the instance score of all keys in that group to SCORE\_G. In Figure 6b,  $\{c\}$  has the highest group score (SCORE\_G=99), so  $c$  is assigned the instance score of 99. We then score the remaining keys of the transaction assuming that keys in the highest-scoring group will be cached.

In subsequent iterations, our algorithm finds the highest-scoring complete group that is a *superset* of all keys that

have been assigned instance scores. In Figure 6b, having scored  $c$ , the highest-scoring complete group that subsumes  $c$  is  $\{b,c,d\}$ , with a group score of 19.3. The unscored keys ( $b,d$ ) are then assigned the score of this complete group (19.3). Intuitively, this is the next set of keys that should be retained assuming that the highest-scoring complete group is already in cache. Our algorithm captures the fact that, once  $c$  is cached,  $d$  should only be cached when  $b$  is cached. The low score of  $b$  contaminates  $d$  but should not contaminate  $c$  (since  $c$  by itself can lead to a transactional hit).

The iterative process described above is repeated until all keys are scored. For our example, the next highest-scoring complete group that is a superset of  $\{b,c,d\}$  is  $\{a,b,c,d\}$ , with a group score of 0.75, which is assigned to key  $a$ , completing the scoring protocol for Figure 6b. Note that all keys will eventually be scored by this algorithm, since they are all part of the trivial complete group containing every key in the transaction.

#### 4.2.3 Scoring Across Transactions

Finally, we describe how to integrate instance key scores across multiple transactions into an *aggregate* value. This final score will be used by the system to decide which keys to evict from the cache. We adopt the following formula:

$$\text{SCORE\_K}(key) = \frac{TS_{key}}{F_{key}} + A_{global}$$

$TS_{key}$  is the sum of all instance scores from Section 4.2.2 across all transactions accessing this key.  $F_{key}$  is the frequency of this key.  $A_{global}$  is the global aging factor.

**Averaging instance scores.** To combine instance key scores into a single value for a given key, we take the running average of these scores. Each time a key is accessed, we add its instance score to the total score  $TS_{key}$  and increment  $F_{key}$  before calculating a new aggregate score. Figure 6e gives the key scores of  $a,b,c,d$  after the execution of the transaction in Figure 6a, assuming that the aging factor is initialized to 0, key size is 1, and the previous  $TS_{key}$  values are 0, 30, 200, and 70 respectively. For example,  $c$  has an instance score of 99 (Figure 6b) for the transaction in Figure 6a, a previous  $TS_{key}$  of 200, and frequency of 99, giving  $\text{SCORE\_K}(c) =$

$\frac{200+99}{99} + 0 = 3.02$  in Figure 6e. Taking the average allows us to account for contamination between different groups.

**Recency.** To account for shifts in object access distributions over time, GDSF, along with other algorithms [8], uses an aging factor to capture object recency. Since previously popular objects can remain in the cache for extended periods of time (due to their high frequencies) and prevent newly popular objects from being stored, the scores of more recent objects should be higher than those of older objects. Towards this end, GDSF applies  $A_{global}$ , a global value that is added to the score of a key upon each access to increase the scores of more recently accessed objects and age older objects out of cache. The value of  $A_{global}$  is updated each time an object is evicted and set as that object’s score. Thus, the factor increases monotonically and ensures that all accesses after this eviction will have scores higher than the last evicted key. In essence, this factor acts as a “reset” on key scores. In Figure 6,  $a$  is evicted after the transaction in Figure 6a executes, and  $A_{global}$  is set to  $a$ ’s score (0.75). This value is then added to  $SCORE\_K$  for each key accessed in the subsequent transaction (Figure 6c). For example,  $c$  has an instance score of 15 (Figure 6d), a previous  $TS_{key}$  of 299, frequency of 100, and  $A_{global}$  of 0.75, giving  $c$  an aggregate key score of  $SCORE\_K(c) = \frac{299+15}{100} + 0.75 = 3.89$  in Figure 6f.

## 5 Optimizations

While our current approach precisely captures the cacheability of each group, it can be prohibitively expensive when the number of complete groups is exponential for some transaction topologies. We address this problem in two ways. First, we observe that many complete groups capture redundant information and introduce *interchangeable groups* to avoid scoring all complete groups, reducing run time overhead. Second, we present a restricted form of grouping, *levels*, that dynamically approximates groups at run time. This technique also enables us to score keys when we do not have access to transaction code (i.e., we do not know the transaction execution graphs).

### 5.1 Interchangeability

We find that the number of complete groups can be exponential with respect to transaction size, even for simple topologies. For example, the TPC-C *Order-Status* transaction in Figure 7a has a depth of three, and the number of complete groups for this transaction is exponential with respect to its depth:  $\{c\}$ ,  $\{o\}$ ,  $\{ol_1, ol_2\}$ ,  $\{c, o\}$ ,  $\{o, ol_1, ol_2\}$ ,  $\{c, ol_1, ol_2\}$ ,  $\{c, o, ol_1, ol_2\}$  make up  $2^3 - 1 = 7$  complete groups.

We observe that transactions often contain complete groups that differ by only a single key. For instance, for every group in which  $c$  is present in Figure 7a, there exists an identical group in which  $o$  replaces  $c$  (and vice-versa). In effect, these keys can be “swapped” with each other and still produce a

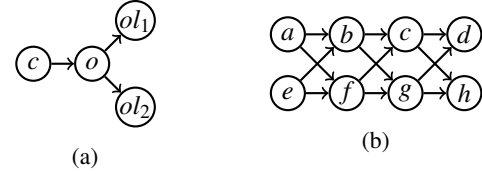


Figure 7: Transactions to demonstrate interchangeability. Figure 7a is a TPC-C *Order-Status* transaction.

complete group. This interchangeability property is powerful: if two keys can be exchanged in any complete group, then deciding to cache one key over the other is entirely dependent on the *individual* scores of these keys, as all other parameters are shared. Consequently, we do not need to calculate the scores of each their complete groups in order to score each key. Consider the groups  $\{c, ol_1, ol_2\}$  and  $\{o, ol_1, ol_2\}$  for the TPC-C *Order-Status* transaction in Figure 7a, assuming  $c$  has a higher individual score than  $o$ . Since  $c$  and  $o$  are interchangeable, we know that  $\{c, ol_1, ol_2\}$  must have a higher group score than  $\{o, ol_1, ol_2\}$ , as all other parameters (the scores of  $ol_1$  and  $ol_2$ ) are shared. Our scoring algorithm favors caching groups with higher scores, so we can avoid calculating the score of  $\{o, ol_1, ol_2\}$  at run time while determining the score for  $o$ .

We can further generalize the idea of interchangeability to *sets* of keys that can also be “swapped” with each other. Continuing the example above, the set of keys  $\{ol_1, ol_2\}$  is interchangeable with  $\{c\}$ , because any complete group that contains  $\{ol_1, ol_2\}$  will remain a complete group if  $\{ol_1, ol_2\}$  is swapped with  $\{c\}$ . We call such sets interchangeable groups:

**Definition 9 (Interchangeable groups).** Let  $s_1$  and  $s_2$  be distinct sets of keys in a transaction with execution graph  $G$ . We define  $s_1$  and  $s_2$  to be interchangeable if

- (1)  $\forall$  complete groups  $g_1$  such that  $s_1 \subseteq g_1$  and  $s_2 \cap g_1 = \emptyset$ ,  $g'_1 = g_1 \setminus s_1 \cup s_2$  is also a complete group and  $L(G, g_1) = L(G, g'_1)$ , and
- (2)  $\forall$  complete groups  $g_2$  such that  $s_2 \subseteq g_2$  and  $s_1 \cap g_2 = \emptyset$ ,  $g'_2 = g_2 \setminus s_2 \cup s_1$  is also a complete group and  $L(G, g_2) = L(G, g'_2)$ .

Like complete groups, interchangeable groups of table accesses can be identified at compile time, as seen in Figure 5. Key accesses are mapped to the vertices at run time. Computationally, interchangeability allows us to reduce the number of complete groups that need to be scored. We compress the representation of complete groups and reduce run time complexity of the scoring algorithm as follows, using Figure 7b as a running example:

• (Compile time) **Find** all interchangeable groups of vertices from the set of complete groups. The complete groups are:  $\{a, e\}$ ,  $\{b, f\}$ ,  $\{c, g\}$ ,  $\{d, h\}$ ,  $\{a, e, b, f\}$ ,  $\{c, g, b, f\}$ ,  $\{d, h, b, f\}$ ,  $\{a, e, c, g\}$ ,  $\{a, e, d, h\}$ ,  $\{c, g, d, h\}$ ,  $\{a, e, b, f, c, g\}$ ,  $\{a, e, b, f, d, h\}$ ,  $\{a, e, c, g, d, h\}$ ,  $\{c, g, b, f, d, h\}$ ,  $\{a, e, b, f, c, g, d, h\}$ . Consider replacing  $\{a, e\}$  with  $\{d, h\}$  in any complete group; the resulting group is still complete. Thus,  $\{a, e\}$  and  $\{d, h\}$  are interchangeable. Using the same



logic, we find that  $\{a,e\},\{b,f\},\{c,g\},\{d,h\}$  are all mutually interchangeable.

- (Compile time) **Compress** complete groups. Denote an access to any one of the mutually interchangeable groups— $\{a,e\},\{b,f\},\{c,g\},\{d,h\}$ —as  $[C]$ . For example,  $\{a,e,b,f,d,h\}$  becomes  $[C,C,C]$ . In this particular example, all groups of size four can be written as  $[C,C]$ , groups of size six as  $[C,C,C]$ , and groups of size eight as  $[C,C,C,C]$ . We call these representations compressed groups.

- (Run time) **Score** compressed groups by replacing vertices with individual keys in each group. Recall from Section 4.2.2 that our instance scoring algorithm scores all complete groups before greedily selecting the highest-scoring ones. With interchangeability, we no longer need to score all complete groups. Assume the minimum scores of the following interchangeable groups are:  $\{a,e\} : 1, \{b,f\} : 10, \{c,g\} : 30, \{d,h\} : 50$ . Since we know that  $\{a,e\}$  and  $\{d,h\}$  are interchangeable and that  $\{d,h\}$  has a higher score, for any complete group containing  $\{a,e\}$ , there must be another complete group containing  $\{d,h\}$  that has the same (or higher) score. Applying this intuition, the highest-scoring complete group corresponding to the compressed group  $[C,C]$  must be composed of the highest and second-highest-scoring interchangeable groups,  $\{d,h\}$  and  $\{c,g\}$  respectively.

In this example, interchangeability decreases the number of groups that need to be considered at run time from fifteen to four. Overall, interchangeability drastically reduces the number of complete groups that must be scored, lowering run time overhead.

## 5.2 Levels

For cases when we do not have access to transaction code, we design a simplified protocol to dynamically infer groups. We first define a *level* to be a set of keys in a transaction that are sent to the data store in parallel; a similar definition is used to group tasks to optimize caching for parallel job execution [11]. In practice, many applications batch parallel reads to the caching system, which often provides an explicit API to support these requests [2]. We assume that applications send requests as soon as their logical dependencies are fulfilled. For instance, the transaction in Figure 6a has levels  $\{a\}, \{b,c\}$ , and  $\{d\}$ . We have  $d$  as a standalone level since it can only be requested once the level containing both  $b$  and  $c$  has finished executing.

Levels produce identical results to our previous grouping strategies for transactions in which all keys and groups are interchangeable (e.g., Figures 7a and 7b). Many real-world workloads are comprised of such transactions (including all the ones we evaluate in Section 8). When transactions do *not* have these properties, levels can miss out on performance opportunities since they only capture a subset of all possible complete groups. For example, in Figure 6a,  $b$  and  $c$  are always scored together under levels, lowering  $c$ 's score. To

maximize transactional hits,  $b$  should instead be scored with  $d$  since both are colder keys, and  $c$  should be given a high score because caching just this key is likely to lead to a transactional hit. We measure the tradeoff between different grouping strategies in Section 8.

## 6 Prefetching

Prefetching is a popular technique to reduce the client-perceived latency of requests by caching items before they are requested [10, 24, 44, 45, 90]. We revisit this strategy in the context of transactions and design a new prefetching algorithm that uses logical dependencies to minimize latency.

Our policy leverages conditional probabilities: once key  $a$  is accessed, it may be very likely that key  $b$  will also be requested in the same transaction. Consider for example `GetLinkedAccounts` in Figure 1: the access to a primary account is almost always followed by requests to the same subsidiary accounts. Our prefetching algorithm tracks these correlated accesses and preemptively brings dependent objects into the cache ( $a_2$  and  $a_3$  are requested alongside the read to  $a_1$ ). Specifically, DeToX stores, for every request  $r$ , sets of keys in subsequent accesses that are logically dependent on  $r$ . DeToX also tracks the frequency of each set and preemptively fetches in the most popular set into cache alongside  $r$ . To bound memory overheads, we restrict the number of dependency sets that can be stored per key and set a frequency threshold below which we do not retain prefetching metadata.

## 7 Implementation

In this section, we describe our implementation of DeToX, which consists of 7K lines of Java. We adopt a standard two-tier architecture in which we layer a Redis (7.0) cache on top of a data store (Postgres (12.10) and TiKV (5.4.3) are supported). A shim layer routes requests, manages concurrency control, and enables prefetching.

### 7.1 Shim Layer

All client requests are directed to our shim layer, which mediates accesses to the cache and data store to support serializable transactions. Read requests go first to Redis. In the absence of a cache hit, the shim forwards the request to the data store and updates the cache with the result. All writes are sent directly to the data store. While our shim layer currently supports a key-value API, we can convert SQL queries to this format, as previous systems have done [34, 35, 47, 58–61, 65–67, 78, 81–84, 92, 98]. We choose to implement a stand-alone shim layer since there is limited open-source support for concurrency control between caching systems and data stores [9, 43, 45, 46, 75, 76, 85, 88]. Furthermore, our shim layer allows us to

easily plug in different systems. We will explore integrating transactional caching directly into systems in future work.

**Concurrency control.** We implement two-phase locking [22] with timeout-based deadlock detection in the shim layer to ensure serializability. The system maintains the following invariant: values in the cache will either 1) reflect the value committed in the (serializable) data store or 2) be protected by an exclusive write lock.

To achieve this, the shim acquires locks on individual objects before sending requests to either storage system. Writes are buffered at the shim layer until commit. Once values are committed in the data store, they are updated in the cache before write locks are released. To handle crashes, we rely on the data store as the source of truth, similar to previous work [46, 75, 76], and we clear the cache after failures to prevent stale reads. We view applying transaction caching to multiversioned systems as a promising avenue for future work.

**Extracting transaction types and execution graphs.** We leverage prior work [36, 94] to obtain transaction execution graphs with table accesses from application code. The widespread adoption of JDBC-style drivers presents a common interface for extracting transactions across applications.

## 7.2 Eviction

Our eviction policy scores keys as a function of their groups as well as their frequency, size, and recency. The latter three are all features that are already available in Redis, which natively supports LRU and LFU. We reuse these metrics to minimize code changes when implementing our algorithm. We make two primary modifications to Redis: we add (1) a global aging factor that is updated during eviction (as detailed in Section 4.2.3) and (2) support for scoring groups of keys. Specifically, we modify the existing method Redis provides for fetching multiple objects to delineate which keys are accessed together. We update key scores only after a transaction has completed so that we have sufficient information to calculate all group scores. Our changes involve less than 100 lines of code and suggest that DeToX can be easily integrated into any caching system. We also implement a trace-driven simulator in Python to evaluate the offline Belady and Transactional Belady algorithms.

## 8 Evaluation

In this section, we evaluate DeToX against existing caching policies on a range of different workloads. Specifically, we aim to answer the following questions:

- How does DeToX compare to single-object algorithms in terms of transactional hit rate and cache efficiency?
- What is the impact of our grouping techniques?
- What is the tradeoff between optimizing for object hit rate and transactional hit rate?

## 8.1 Experimental Setup

We run our shim layer and Postgres on separate c5a.4xlarge Amazon EC2 instances (16 CPUs, 32GB RAM) and use a memory-optimized r5.4xlarge machine (16 CPUs, 128GB RAM) for Redis. Clients run on c5a.16xlarge instances (64 CPUs, 128GB RAM). We host all machines in the same region with low network latency (0.2ms). For our experiments, we report the average of three 5-minute runs with 60 seconds of warm-up time. When an eviction is needed, we score 10 random samples and choose one to evict among these candidates. This strategy removes the overhead of maintaining a sorted list of keys without degrading performance and is popular in many caching systems [2, 79], including Redis.

**Benchmarks.** We evaluate DeToX against single-object baselines as well as the policies developed in PACMan [11] (their LFU-F is equivalent to our LFU; we evaluate their LIFE algorithm) and ChronoCache [45], a state-of-the-art prefetching system that leverages transactional dependencies. We measure performance on a range of workloads. **TAOBench** [26] is an open-source social network benchmark based on Meta’s production traces. We run the Product Group 1, 2, and 3 workloads, which represent distinct sets of (anonymized) applications at Meta that share data and use the same product infrastructure. All workloads are read-heavy and skewed, typical of most social networks. They contain point reads and writes (inserts, updates, and deletes) as well as read-only and write-only transactions. All transactions are “flat” (they contain no logical dependencies). Since transaction code is not available for this benchmark, we use levels to score groups for eviction.<sup>1</sup> We run experiments with 100M objects for a total data size of around 1 TB. **Epinions** [37] consists of nine transaction types that represent behavior observed on a consumer reviews website. We run the benchmark with 2M user and 1M items for a total data size of roughly 1 TB. **SmallBank** [87] contains six types of transactions that model a simple banking application. We configure it to run with 500M (uniformly accessed) accounts (total size of 1 TB). **TPC-C** [33], a standard OLTP benchmark, simulates the business logic of e-commerce suppliers with five types of transactions. We configure TPC-C to run with 100 warehouses (total size of 8GB). In line with prior transactional key-value stores [34, 81], we use a separate table as a secondary index on the Order table to locate a customer’s latest order in the Order-Status transaction, and on the Customer table to look up customers by their last names (for the Order-Status and Payment transactions).

## 8.2 Application Benchmark Results

We show THR over different cache sizes for all benchmark workloads in Figures 8 and 10. We omit some throughput and

<sup>1</sup>TAOBench [26] chooses to model workloads using probability distributions rather than fixed query types for adaptability.

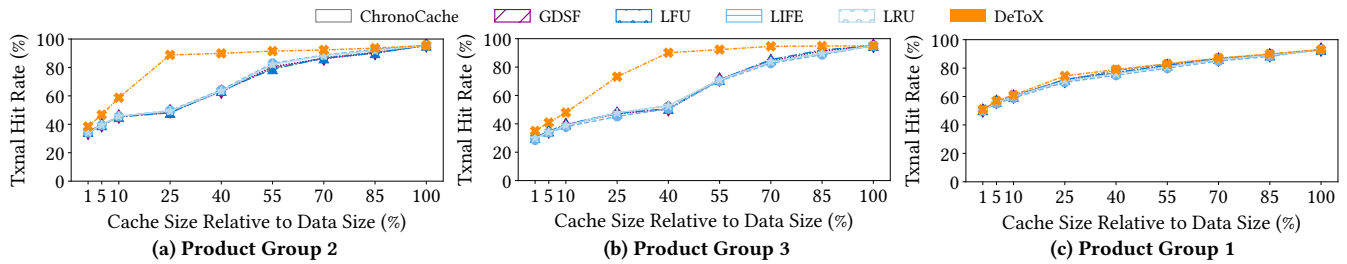


Figure 8: TAOBench THR results.

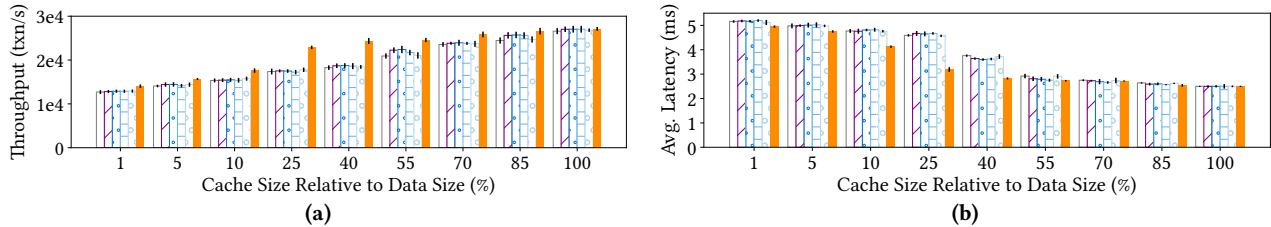


Figure 9: TAOBench PG2 results.

latency graphs for space but describe results in text. Since all transactions in these workloads have symmetric structures, there is no difference in performance between our various grouping techniques. We detail the tradeoffs between these optimizations in the next section.

**TAOBench.** DeToX obtains up to 76% higher transactional hit rates on the TAOBench PG2 and PG3 workloads compared to single-object caching algorithms (Figures 8a and 8b). DeToX achieves this with better cache efficiency: at the 25% cache size relative to data size (a common setup following the “80-20 rule”), the protocol achieves an 88% transactional hit rate while the best single-object algorithm requires 3.4x more cache space to attain the same result on PG2. Results are similar for PG3 for which the system requires a 2.2x smaller cache. Throughput increases by 31% (from 18K txns/s to 24 txns/s) for PG2 and 30% for PG3 (from 31K txns/s to 40 txns/s), while latency decreases by 30% (4.6ms to 3.2ms) for PG2 (Figure 9b) and 29% for PG3 (2.3ms to 1.6ms).

PG2 is read-dominant (>96%) with a mix of point reads, short transactions (<10 operations), and larger read transactions that span up to 40 keys. The point reads and shorter transactions make up 60% of the workload and largely access a small group of hot keys. Consequently, all algorithms achieve a THR of over 45% for small cache sizes (10% relative size). The longer read transactions follow one of two patterns: transactions access either a combination of hot and warm keys (25%), or hot and cold keys (11%). Transactions from the first category are more beneficial to cache since their keys are more frequently accessed and more likely to lead to transactional hits. There is little benefit in caching any of the keys in the second category since the cold keys contaminate all the other ones.

Under DeToX, the cache initially chooses to cache keys that belong to transactions in the first category. Thus, transactional hit rate improves as the cache size increases from 10% to

40% (Figure 8a). Past this point, the cache begins to retain more keys from transactions in the second category, but the performance benefit is limited since these requests rarely lead to transactional hits. In contrast, single-object algorithms use only individual object features to score keys, so they retain hot keys from transactions in both categories. Transactional hit rate increases slowly up to the 55% cache size at which point the cache becomes large enough to begin storing the warm keys from the first transaction category. Since the TAOBench workloads have no temporal patterns, GDSF and LFU provide slightly higher hit rates compared to LRU for all cache sizes. While LIFE uses levels, it performs poorly because it only uses the size of levels to make eviction decisions.

Similarly, in PG3, DeToX achieves better cache efficiency by not retaining contaminated keys. This workload has a smaller portion of point reads and shorter transactions (50%), so hit rates at smaller cache sizes are lower for all policies. Longer read transactions span up to 60 items and also fall into two categories. There are more transactions in the first category (33% compared to 25% in Product Group 2), so transactional hit rates grow more slowly with respect to cache size since more warm keys need to be cached.

In contrast to the other workloads, PG1 (Figure 8c) consists mainly of point reads and some short read transactions (of size four or smaller), which together make up over 97% of all requests. Our algorithm does not improve transactional hit rate over single-object policies because most hits result from standalone requests and short read transactions to a set of highly popular keys, which single-object algorithms already cache effectively. Throughput increases by 2% (from 82K txn/s to 84K txn/s), and latency decreases by 2% (from 0.61ms to 0.60ms).

ChronoCache has similar hit rates to single-object algorithms since there are no dependencies within transactions for this benchmark; the results simply reflect its eviction policy, LRU. The middleware layer, which does dependency

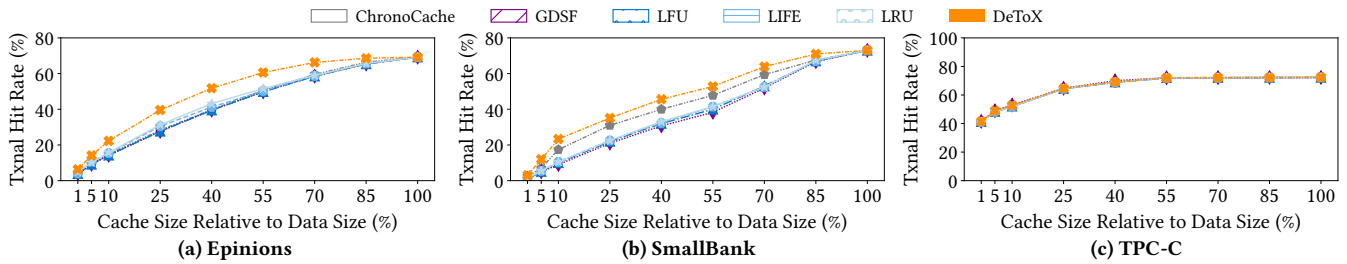


Figure 10: OLTP benchmark THR results.

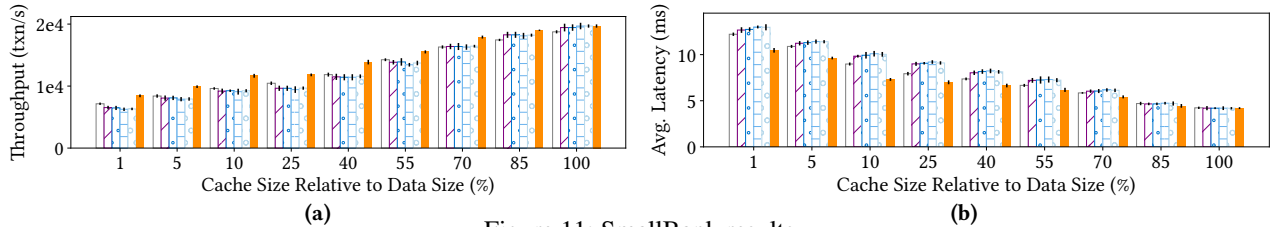


Figure 11: SmallBank results.

analysis at run time, quickly becomes the bottleneck.

**Epinions.** Epinions centers around user interactions and item reviews. It contains five read-only transactions and four update transactions. Users have both an n-to-m relationship with items (i.e., representing user reviews and ratings of items) and an n-to-m relationship with other users. There are no logical dependencies in the transactions of this workload (all operations can be parallelized).

DeToX provides up to 41% increase in transactional hit rate (Figure 10a), translating into 29% improvement in throughput (from 12K txn/s to 17K txn/s) and 25% decrease in latency (from 6.9ms to 5.5ms). At the 25% cache size, DeToX is 1.6x more efficient than the other algorithms. The transactions in Epinions request some group of objects related to a particular user or item (e.g., get all the reviews from one user), so our policy is able to successfully capture the n-to-m relationships in the data with its scoring mechanism. In contrast, the single object policies focus on caching individually popular keys without taking into account correlation between accesses. Since there are no dependencies between or within transactions for this workload, ChronoCache is unable to successfully prefetch objects.

**SmallBank.** SmallBank consists of requests to the Accounts, Checking, and Savings tables with six transaction types. Its transactions are relatively small, involving four distinct keys at most. Roughly two-thirds of operations are reads. Each customer account is materialized as three separate entries in each table and is accessed with a uniform distribution. There is high correlation between accesses to a customer’s row in the Accounts table and the customer’s rows in the other two tables.

Our algorithm provides up to a 1.3x increase in transactional hit rate (Figure 10b). The absolute hit rates remain relatively low for smaller cache sizes because of the uniform access distribution to customer accounts. Transactional hit rate increases linearly for all algorithms since more cache space directly

results in more hits. DeToX is 1.6x more efficient than the next best-performing algorithm at the 25% cache size.

We observe up to a 28% increase in throughput (from 12K txn/s to 16K txn/s) and 26% decrease in latency (from 6.8ms to 5.4ms) on this workload (Figures 11a and 11b). The long tail in access patterns and short transactions of this workload limit the benefits of our eviction algorithm over single-object alternatives, which all have similar performance.

For this workload, around two-thirds of performance improvement can be attributed to prefetching. We compare our eviction algorithm without prefetching (DeToX-E), LRU with prefetching (LRU-P), and our full policy (DeToX). DeToX-E increases throughput by 9%, LRU-P by 19%, and DeToX by 28% (graph omitted for space).

**TPC-C.** TPC-C is notably write-heavy and has transactions that can span over 50 items. Its requests tend to fall into two categories: either they access a small set of popular keys (i.e., those in the Warehouse and District tables) or a larger range of keys from a distribution with a long tail (Customer, Item, Stock). Single-object caching algorithms are designed to cache the former while the latter almost always results in transactional misses. For instance, *New-Order* accesses a key in each of the Warehouse, District, and Customer tables before requesting 10 to 15 items from the Item and Stock tables, which are chosen from a skewed distribution.

Consequently, TPC-C cannot benefit from transactional caching: most transactions access a small set of hot keys that are already in the cache (the object hit rate is >50% with a 10% cache size in Figure 10c) along with a larger set of cold keys that are unlikely to be cached and contaminate the other keys (hit rate grows slowly as cache size increases). Moreover, transactions tend to access keys in quick succession (e.g., once an order is placed, it is then processed, paid for, and delivered), so recency is especially important in this workload. All algorithms incorporate recency in some form,



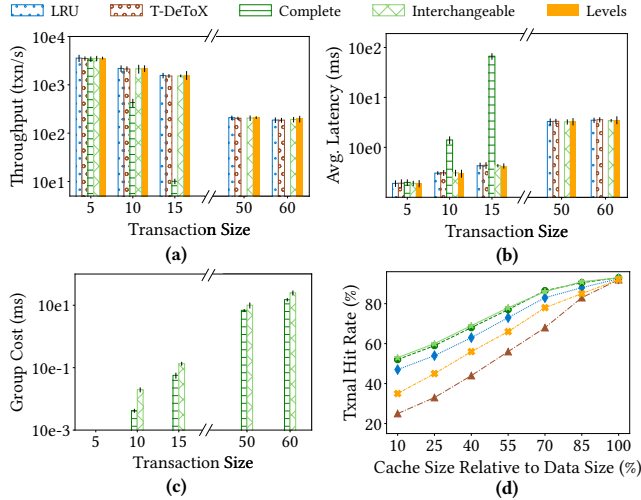


Figure 12: (a),(b),(c). Microbench. 1 (d) Microbench. 2.

so performance is similar across these policies, with up to 9K txns/s and 27ms avg. latency. DeToX performs on par with single-object policies.

### 8.3 The Need for Dependency Analysis

In this section, we investigate the relative merits of our grouping optimizations. The dependency analysis required for complete groups can impose overheads in two ways: (1) the cost of updating the scores of each key in each group and (2) metadata overhead associated with scoring. Interchangeability can reduce the number of groups that need to be scored, leading to better performance. On the other hand, levels discount unbalanced topologies while T-DeToX, a baseline that scores all keys of a transaction together, ignores dependencies. These simpler policies reduce overhead in some cases but restrict the groups that keys can belong to, leading to worse performance.

**Performance impact.** Microbenchmark 1 intentionally captures the worst-case scenario for grouping. We run a single transaction type with the topology in Figure 6a, and we extend the right branch of the graph for larger transaction sizes. Each read uniformly accesses keys at random among 10M objects. We measure throughput and latency as we increase transaction size up to 60 (equivalent to the largest transactions in the TAOBench workloads). Figures 12a and 12b show that performance for complete groups decreases dramatically as transaction size increases due to the exponential number of complete groups: for a transaction of size 15, over 16K groups have to be scored. Note that the bars for throughput and latency are omitted for complete groups for transaction sizes greater than 15 since these experiments did not finish in reasonable amount of time. In contrast, performance degradation is minimal with interchangeable groups (<5% difference compared to LRU at size 60). There are only a linear number of groups that must be scored with respect to transaction size since all keys in the right branch of this topology are interchangeable. Finally, levels offer similar performance to

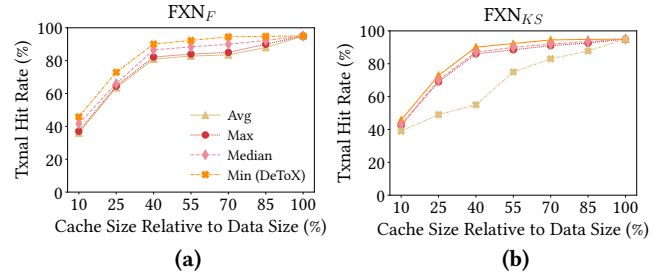


Figure 13: Scoring heuristics

LRU. Each key can only belong to one level per transaction, so larger transaction sizes do not increase overhead. The run time CPU overhead of both interchangeable groups and levels is within 5% of that of single-object algorithms for all microbenchmarks and previous benchmark workloads.

Moreover, the one-off cost of finding complete and interchangeable groups at compile time remains low: transactions of size 60 (with 100K+ groups due to worst-case topologies) require less than five minutes to process (Figure 12c). All benchmark workloads require less than 30 seconds for dependency analysis.

While dependency analysis incurs a static cost, it can lead to significant benefits compared to more basic forms of grouping (levels and T-DeToX), which ignore some or all dependency information. Microbenchmark 2 quantifies the worst-case scenarios for levels and T-DeToX. We run a single transaction type with the topology in Figure 6a in which the keys in vertices *a* and *c* are hot keys chosen from a Zipfian distribution while keys in *b* and *d* are cold keys chosen from a uniform distribution over 10M objects. Using levels causes keys in *b* and *c* to be scored together. However, keys in *b* are rarely accessed, and contaminate keys in *c*. T-DeToX makes even worse eviction decisions since it scores all keys in *a*, *b*, *c*, and *d* together. Using complete and interchangeable groups would instead cause keys in *b* and *d* to be scored together, enabling the algorithm to capture the fact that caching *c* individually reduces critical length. We find that complete and interchangeable groups significantly outperform levels (53% increase) and T-DeToX (139% increase) for THR (Figure 12d). Complete and interchangeable groups offer similar performance to LRU since these policies cache keys in *c*, which are frequently accessed.

**Memory overheads.** Metadata overhead in DeToX is low. Our algorithm stores two additional counters (total group score, individual score) per key and a global aging factor for eviction. While prefetching, DeToX stores dependency sets. On TAOBench, additional metadata takes up less than 1% of the cache space. For workloads in which prefetching is more prevalent, metadata overheads increase slightly. For example, in SmallBank, additional metadata grows to 2%. DeToX must store the dependency set associated with each transaction (1.5 keys on average).

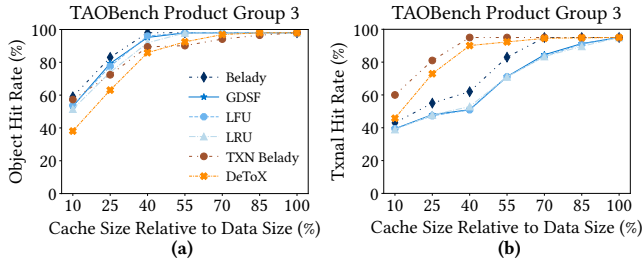


Figure 14: Object (a) and transactional (b) hit rates.

## 8.4 Scoring Heuristics

We evaluate different heuristics for calculating instance ( $F_{XNF}$ ) and aggregate scores ( $F_{XNKS}$ ). DeToX uses the *minimum* frequency of keys in a group for the instance score, and *averages* instance scores to compute an aggregate score (Section 4.2). We measure transactional hit rates for simple functions (average, maximum, median, minimum) in Figure 13 for the PG3 workload (results are similar across workloads).

For assigning key instance scores, we find that, as expected, Min provides the best performance (Figure 13a). Since we only get a transactional hit if *all* keys of a group are cached, the key with the smallest frequency should have outsized impact on the group score. The other functions discount this information and thus perform worse. However, these functions still encode the all-or-nothing property of transactions to some extent since they assign the same instance scores to all keys in a particular group. As a result, we still observe higher hit rates than single-object policies.

Average and Median are the most effective functions for calculating aggregate key score (Figure 13b). Max yields a lower hit rate since it assigns each key the score of its highest-scoring group, but this may not be the most frequent group that contains this key. Min provides markedly lower performance (up to 64% lower hit rates). Each key is assigned the score of its lowest-scoring group, so most scores converge to the lowest group score (the smallest frequency of any key). As a result, most scores are low and do not differ by much.

## 8.5 OHR versus THR

There is a tradeoff between optimizing for latency and for system load. Figure 14 shows the OHR and THR of online algorithms as well as Belady and Transactional Belady (see Appendix A). As expected, Belady outperforms other algorithms for object hit rate. Conversely, DeToX and Transactional Belady give some of the lowest object hit rates. However, these two algorithms significantly outperform the other policies for transactional hit rate (and result in better throughput and latency as shown in Section 8.2). While we focus on PG3 here, we find similar results on the other workloads (omitted due to lack of space).

The difference between OHR and THR illustrates a tradeoff between reducing I/O bandwidth and optimizing for latency.

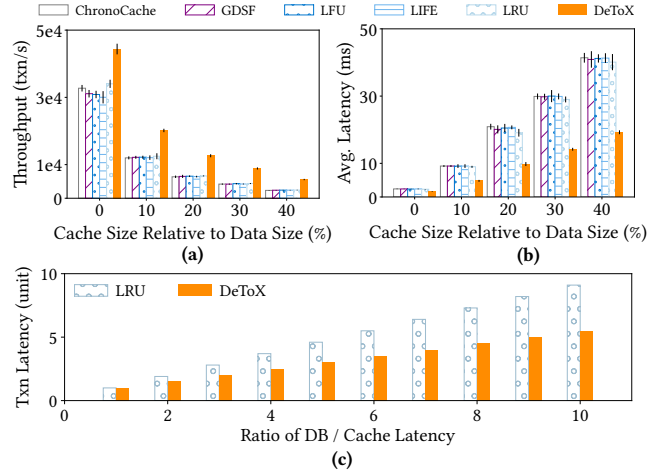


Figure 15: Network latency (a), (b) and simulation (c) results.

OHR prioritizes the absolute number of requests that can be served from cache, minimizing requests to disk. In contrast, THR focuses on the number of latency reductions for transactions, leading to lower latency and higher throughput. There are practical motivations for choosing THR as the caching objective: with increasing elasticity from cloud resources, applications often focus on latency optimization for which large wins are possible with DeToX.

## 8.6 Transactional Hit Rate

Transactional hit rate is independent of system specifics; only relative throughput and latency gains differ when cache / system latency changes. We confirm this by (1) varying this ratio (both experimentally and through simulation) and (2) evaluating DeToX with an alternative key-value store, TiKV [4].

**Network latency.** We inject latency between the shim layer and data store to simulate scenarios in which the latter is hosted in a remote cloud region. Figure 15 shows that the performance improvement with DeToX grows as network latency increases. With no additional network latency (0ms), there is a 30% increase in throughput and 29% decrease in latency between DeToX and the best single-object policy for PG3. With a WAN delay of 10ms, there is a 61% increase in throughput and 47% decrease in latency.

**Simulation results.** To illustrate the impact of cache and data store request times, we provide results for the TAOBench PG2 workload. At the 25% cache size, the THR for this workload is around 90% for DeToX and 50% for the other policies (Section 8.2). We vary request times for the cache and the data store (DB), using arbitrary units to represent latency. As we increase the ratio of DB to cache latency in Figure 15c, we find that the difference in request latency between LRU and DeToX increases from 0% to 65% as request times to the data store lengthen.

**Transactional key-value store.** We confirm that both the difference in transactional hit rate and gains in cache

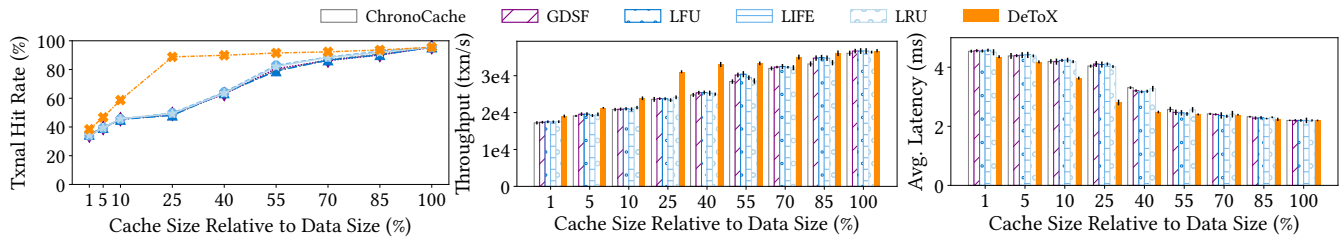


Figure 16: TAOBench PG2 results on TiKV.

efficiency (3.4x) remains identical when executing atop TiKV, demonstrating that these metrics are independent of the setup chosen (Figure 16). In contrast, as TiKV exhibits higher throughput and lower latency than Postgres, throughput and latency gains fall to 19% and 15% respectively.

## 9 Related Work

**Eviction.** There is a wide range of research on single-object caching policies that consider frequency [23, 40, 41, 53, 63], recency [32, 42, 52, 70], the number of unique keys between accesses [14, 50, 57, 64, 73], the variable sizes of objects [5, 25], and combinations of these features [6, 7, 12, 13, 15, 18, 20, 28, 49, 51, 56, 77, 80, 99]. Some specialized eviction policies optimize for flash storage [74], adapt to changing workloads [17, 19, 21, 30, 31, 39, 89], or consider network bandwidth and download time for proxy caches [93]. These previous efforts do not explicitly address how caching should be optimized for parallel accesses in transactions.

PACMan [11] presents eviction algorithms targeted towards job processing based on the all-or-nothing property: for jobs that issue tasks in parallel, latency only improves if all parallel tasks are cached. Similarly, existing literature on web caching [7, 18, 91] focuses on maximizing the *page* hit rate since latency is reduced only when all parts of a page are cached. Transactional hits in DeToX are based on a similar insight. However, DeToX addresses the issue of complex, unbalanced dependency graphs and recognizes that keys can be shared across many transactions.

**Admission algorithms.** In contrast to eviction algorithms, admission policies decide what to *allow* into the cache by enforcing a threshold based on object scores. These algorithms have often been applied alongside eviction policies [7, 21, 40, 52, 62]. While we focus on eviction and prefetching in this paper, our grouping and scoring strategies can feasibly extend to admission, which we will explore in future work.

**Prefetching.** Prefetching has been applied extensively to web caching [10, 90]. Past work focuses on web page analysis [38, 55, 68, 71, 86, 95, 96], which most stand-alone caches do not support [2, 69]. Other research [24, 44, 45, 72] centers around reducing the latency of query execution using dependency analysis. These works assume that each client issues queries sequentially, so any cache hit can

improve latency. Instead, DeToX caches in order to maximize transactional hit rate. Furthermore, none of these systems provide isolation guarantees or consider how eviction policies should be modified to handle transactions.

**Cache coherence.** Previous work combining transactions and caching focuses on maintaining isolation guarantees for cache coherence [1, 54, 76, 97]. In contrast, we focus on what objects to cache for performance. DeToX ensures serializability while optimizing for transactional hit rate.

## 10 Conclusion

In this paper, we study the problem of transactional caching. Standard caching policies fail to account for the all-or-nothing property of transactions, resulting in inefficient choices for which objects to retain in cache. In light of this issue, we provide a formal framework to quantify the latency impact of caching for transactions and introduce transactional hit rate as the key metric for this setting. We then present DeToX, a novel caching system targeting at transactional workloads. DeToX maximizes transactional hit rate by centering its caching policy around scoring groups of keys together. We consider how keys are accessed in parallel through complete groups and introduce interchangeable keys as an optimization to reduce the overhead of having to score many groups at run time. We also describe levels as a technique for cases when transaction code is not available. Our implementation is lightweight and deployable on a range of existing caching systems and data stores. DeToX improves THR by up to 1.3x and cache efficiency by up to 3.4x. This work demonstrates that many applications can benefit measurably from transactional caching.

## Acknowledgements

We thank Matt Burke, our anonymous reviewers, and our shepherd Wyatt Lloyd for their insightful feedback as well as Akshay Ravor for his engineering contributions. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF GRFP Award DGE-1752814, a Meta Next-Generation Infrastructure award, and gifts from Amazon, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, and VMWare.

## References

- [1] Amazon elasticache, November 2021.
- [2] Redis, February 2021.
- [3] Postgresql, 2022.
- [4] Tikv, 2022.
- [5] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, page 293–305, New York, NY, USA, 1996. Association for Computing Machinery.
- [6] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A Fox. Caching proxies: Limitations and potentials. 1995.
- [7] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. 11(1):94–107, jan 1999.
- [8] Jose Aguilar and Ernst L. Leiss. A web proxy cache coherency and replacement approach. In *Proceedings of the First Asia-Pacific Conference on Web Intelligence: Research and Development*, WI '01, page 75–84, Berlin, Heidelberg, 2001. Springer-Verlag.
- [9] Marcos K. Aguilera, Joshua B. Leners, and Michael Wal-fish. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 245–262, New York, NY, USA, 2015. Association for Computing Machinery.
- [10] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl.*, 3(1):18–44, 2011.
- [11] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 20, USA, 2012. USENIX Association.
- [12] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, mar 2000.
- [13] Hyokyung Bahn, Kern Koh, S.H. Noh, and S.M. Lyul. Efficient replacement of nonuniform objects in web caches. *Computer*, 35(6):65–73, 2002.
- [14] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.
- [15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.
- [16] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [17] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018.
- [18] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), jun 2018.
- [19] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.
- [20] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.
- [21] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, March 2017. USENIX Association.
- [22] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [23] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.
- [24] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. In *Proceedings of the 2004 ACM SIGMOD international*



conference on Management of data - SIGMOD '04. ACM Press, 2004.

- [25] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, 1997.
- [26] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. Taobench: An end-to-end benchmark for social network workloads. *Proceedings of the VLDB Endowment*, 15(12):1965–1977, 2022.
- [27] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, 1998.
- [28] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*, pages 114–123. Springer, 2001.
- [29] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, March 2011.
- [30] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.
- [31] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.
- [32] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.
- [33] The Transaction Processing Performance Council. Tpc-c, 2021.
- [34] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18*, page 727–743, USA, 2018. USENIX Association.
- [35] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, June 2016.
- [36] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17*, page 235–250, New York, NY, USA, 2017. Association for Computing Machinery.
- [37] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. volume 7, pages 277–288. VLDB Endowment, 2013.
- [38] Josep Domenech, Jose A. Gil, Julio Sahuquillo, and Ana Pont. Using current web page structure to improve prefetching performance. *Comput. Netw.*, 54(9):1404–1417, jun 2010.
- [39] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.
- [40] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.
- [41] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11*, New York, NY, USA, 2011. Association for Computing Machinery.
- [42] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 43(1):123–136, jun 2015.
- [43] Shahram Ghandeharizadeh, Jason Yap, and Hieu Nguyen. Strong consistency in cache augmented SQL systems. In *Proceedings of the 15th International Middleware Conference on - Middleware '14*. ACM Press, 2014.
- [44] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Scott Foggo, and Anil Pacaci. Apollo: Learning query correlations for predictive caching in geo-distributed systems, 2018.
- [45] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, page 2391–2406, New York, NY, USA, 2020. Association for Computing Machinery.

- [46] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for orms. In *Proceedings of the 12th International Middleware Conference*, Middleware '11, page 320–339, Laxenburg, AUT, 2011. International Federation for Information Processing.
- [47] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. ACM, October 2021.
- [48] Saied Hosseini-Khayat. *Investigation of Generalized Caching*. PhD thesis, USA, 1998. UMI Order No. GAX98-07761.
- [49] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.
- [50] Song Jiang, Feng Chen, and Xiaodong Zhang. Clockpro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.
- [51] Shudong Jin and Azer Bestavros. Greedydual\* web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.
- [52] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94*, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [53] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.
- [54] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. Easy freshness with pequod cache joins. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14*, page 415–428, USA, 2014. USENIX Association.
- [55] Bin Lan, Stephane Bressan, Beng Chin Ooi, and Kian-Lee Tan. Rule-assisted prefetching in web-server caching. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, page 504–511, New York, NY, USA, 2000. Association for Computing Machinery.
- [56] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143, 1999.
- [57] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.
- [58] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, October 2017.
- [59] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, Lombard, IL, April 2013. USENIX Association.
- [60] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and Latency-Optimal Read-Only transactions. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 135–150, Savannah, GA, November 2016. USENIX Association.
- [61] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-Optimal Read-Only transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 333–349. USENIX Association, November 2020.
- [62] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, jul 2015.
- [63] Dhruv Matani, Ketan Shah, and Anirban Mitra. An o(1) algorithm for implementing the lfu cache eviction scheme. *arXiv preprint arXiv:2110.11602*, 2021.
- [64] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.
- [65] Syed Akbar Mehdi, Cody Littlely, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency

- with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, Boston, MA, March 2017. USENIX Association.
- [66] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014. USENIX Association.
- [67] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, Savannah, GA, November 2016. USENIX Association.
- [68] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.
- [69] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.
- [70] Elizabeth J O’neil, Patrick E O’neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.
- [71] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, jul 1996.
- [72] Mark Palmer and Stanley B Zdonik. *Fido: A cache that learns to fetch*. Brown University, Department of Computer Science, 1991.
- [73] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.
- [74] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cfrru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES ’06*, page 234–241, New York, NY, USA, 2006. Association for Computing Machinery.
- [75] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic si-cache: Consistent and scalable caching in multi-tier architectures. *The VLDB Journal*, 20(6):841–865, dec 2011.
- [76] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI’10*, page 279–292, USA, 2010. USENIX Association.
- [77] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on networking*, 8(2):158–170, 2000.
- [78] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, March 2022.
- [79] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.
- [80] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance evaluation*, 46(2-3):125–137, 2001.
- [81] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD ’17*, page 283–297, New York, NY, USA, 2017. Association for Computing Machinery.
- [82] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. ACM, October 2021.
- [83] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, April 2020.

- [84] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional Key-Value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 63–80. USENIX Association, November 2020.
- [85] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22*, page 4–18, New York, NY, USA, 2022. Association for Computing Machinery.
- [86] Na Tang and V. Rao Vemuri. An artificial immune system approach to document clustering. In *Proceedings of the 2005 ACM Symposium on Applied Computing, SAC '05*, page 918–922, New York, NY, USA, 2005. Association for Computing Machinery.
- [87] The H-Store team. Smallbank benchmark, 2013.
- [88] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proc. VLDB Endow.*, 11(5):648–662, jan 2018.
- [89] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with {ML-based}{LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
- [90] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, oct 1999.
- [91] Justin Wang, Benjamin Berg, Daniel S. Berger, and Siddhartha Sen. Maximizing page-level cache hit ratios in largeweb services. *SIGMETRICS Perform. Eval. Rev.*, 46(2):91–92, jan 2019.
- [92] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data, ACM*, June 2016.
- [93] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8-13):977–986, 1997.
- [94] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, page 1689–1704, New York, NY, USA, 2016. Association for Computing Machinery.
- [95] Lifang Xu, Hongwei Mo, Kejun Wang, and Na Tang. Document clustering based on modified artificial immune network. In Guo-Ying Wang, James F. Peters, Andrzej Skowron, and Yiyu Yao, editors, *Rough Sets and Knowledge Technology*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [96] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. *KDD '01*, page 473–478, New York, NY, USA, 2001. Association for Computing Machinery.
- [97] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, jun 2018.
- [98] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. volume 35, New York, NY, USA, December 2018. Association for Computing Machinery.
- [99] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

## A Appendix

We prove the optimal offline transactional caching problem is NP-Hard. We begin by providing intuition for how and why traditional optimal offline caching policies fail to translate to transactional caching.

### A.1 Transactional Belady

We straightforwardly adapt Belady’s optimal caching policy [16] to the transactional context by defining *Transactional Belady*, a caching policy that evicts keys that result in *transactional hits* furthest in the future. While this extension is intuitive, it does not offer optimal performance even for flat, uniformly-sized transactions that access equally-sized objects, as we prove below.

Consider the execution trace in Figure 17 with cache capacity of 5. All transactions access three keys, either all from set  $S_1: \{t, u, v, t', u'\}$  or set  $S_2: \{x, y, z, x', y'\}$ .  $T_1$  and  $T_2$  access only keys from the former group, while  $T_3$  and  $T_4$  access only keys from the latter.  $T_5$  and  $T_6$  access keys from  $S_1$  and overlap in  $v$ , while  $T_7, T_8, T_9$  overlap in  $x', y, z$  from  $S_2$ . Transactional



T	Keys accessed	Cache state	Optimal cache state
1	$t, u, v$	-	-
2	$t', u', v$	$t, u, v$	-
3	$x, y, z$	$t, u, v, t', u'$	-
4	$x', y', z$	$t, u, v, t', u'$	$x, y, z$
5	$t, u, v$	$t, u, v, t', u'$	$x, y, z, x', y'$
6	$t', u', v$	$t, u, v, t', u'$	$x, y, z, x', y'$
7	$x, y, z$	$t, u, v, t', u'$	$x, y, z, x', y'$
8	$x', y, z$	$t, u, v, t', u'$	$x, y, z, x', y'$
9	$x', y', z$	$t, u, v, t', u'$	$x, y, z, x', y'$

Figure 17: Non-optimality of Transactional Belady. Red keys indicate ones that lead to transactional hits.

Belady evicts keys that yield a transactional hit furthest in the future. After  $T_3$ 's execution, the algorithm evicts  $x, y, z$  as they would first yield a hit at  $T_7$  while the other keys would lead a hit at  $T_5$  and  $T_6$ . A similar reasoning leads the algorithm to evict  $x', y', z'$  after  $T_4$  executes. This strategy yields two transactional hits (for  $T_5$  and  $T_6$ ). Unfortunately, evicting  $x, y, z$  after  $T_3$  is the wrong decision. Keeping all keys of set  $S_2$  in the cache yields three transactional hits  $T_7, T_8, T_9$ . As a result, Transactional Belady achieves only two transactional hits, while an optimal caching policy would achieve three.

Transactional Belady does not account for shared keys across transactions. It caches set  $S_1$ , which is shared across two transactions, instead of keys in set  $S_2$ , which is shared across three transactions. Belady assumes that a cache hit closer in the future is always as valuable as a cache hit further out. This assumption holds when a single cached object provides a single object hit but breaks down when keys are shared across transactions. In these cases, an equal number of cached objects can produce varying numbers of transactional hits.

## A.2 Optimal Offline Transactional Caching is NP-Hard

We demonstrate that the optimal offline transactional caching problem (TxPolicy) is NP-Hard through a reduction from the variable-sized caching problem, CACHING(FAULT, OPTIONAL), introduced in [29].

We first provide intuition for our reduction. A page hit is only possible if the entire page is present in the cache, regardless of its size. The objective of CACHING(FAULT, OPTIONAL) is to minimize the number of page faults, or the number of pages accessed and missed. We convert each page of size  $X$  into a transaction without dependencies that accesses  $X$  operations. Therefore, there is only a transaction hit when the entire transaction is in the cache. This transforms CACHING(FAULT, OPTIONAL) into an easier version of TxPolicy with two simplifying assumptions: (1) all transactions will use unique keys, so that retaining a key in the cache from any single transaction provides no benefit to any other transaction, and (2) there are no logical dependencies. If an optimal offline transactional

caching policy exists, then through this reduction, we have the optimal policy for CACHING(FAULT, OPTIONAL).

We now formally describe CACHING(FAULT, OPTIONAL) from [29]. CACHING(FAULT, OPTIONAL) asks,

Given a set of pages  $p_1, \dots, p_k$  with sizes

$\text{SIZE}(p_1), \dots, \text{SIZE}(p_k)$ , request sequence  $r_1, \dots, r_m \in \{p_1, \dots, p_k\}$ , cache size  $C$ , and cost bound  $F$ , is there a replacement policy that serves  $r_1, \dots, r_m$  with cache size  $C$  and incurs a total fault cost at most  $F$ ?

A *fault* is incurred when  $r_i \notin C_i$ , where the FAULT parameter states that each fault has cost 1. The OPTIONAL parameter requires that  $\forall i > 1, C_i \subseteq \{C_{i-1} \cup r_i\}$ ; informally, the caching policy does not have to admit the most recent page.

We formally define the offline transactional caching problem, based on our formalisms from Section 3.

**Definition 10** (*Offline transactional caching policy*). An *offline transactional caching policy* is a function  $P$  that takes a sequence of transactions  $T_1, T_2, \dots, T_m$ , cache size  $n$ , and outputs a sequence of cache states  $C_1, C_2, \dots, C_m$ , with the following restrictions:

1.  $C_1 = \emptyset$ .
2.  $\forall i > 1, C_i \subseteq \{C_{i-1} \cup T_{i-1}\}$ .

TxPolicy asks,

Given a set of transactions  $T_1, \dots, T_m$ , cache size  $C$ , is there an offline transactional caching policy that serves  $T_1, \dots, T_m$  with cache size  $C$  and incurs at most  $F$  transactional misses? We define *transactional misses* as the number of  $i$  where  $T_i \not\subseteq C_i$ , or the number of transactions that cannot be served from cache.

**Theorem 1.** *The optimal offline transactional caching problem is NP-Hard.*

*Proof.* We reduce CACHING(FAULT, OPTIONAL) to TxPolicy through the following polynomial-time reductions. Each page  $p_i$  is reduced to a transaction  $T_i$ .  $\text{SIZE}(p_i)$  new tables are created per transaction, each with only one key. Let  $X$  be one such table. A read operation on the sole key of that table  $x \in X$  is inserted into the transaction  $T_i$ . There are no logical dependencies. Cache size  $C$  is preserved. The maximum fault cost  $F$  is converted to the maximum number of transactional misses. If there exists a policy solving the offline transactional caching problem, run it with these parameters. Its output is the output to the CACHING(FAULT, OPTIONAL) problem. CACHING(FAULT, OPTIONAL) is NP-Hard; therefore, the offline transactional caching problem is NP-Hard.  $\square$

## B Artifact Appendix

### Abstract

DeToX is a transactional caching system that leverages insights on transactional hit rate to improve caching performance for transactional workloads. DeToX is implemented as a shim layer that integrates with caching and database systems. In addition to DeToX, the artifact contains several other implementations. First, there is a modified version of ChronoCache, a middleware predictive query caching system, that measures transactional hit rate, integrates with Redis, and supports several benchmarks not available for the original system. There is also a modified version of Redis that supports several eviction algorithms, including DeToX's eviction algorithm and LIFE from the PACMan paper. Finally, there is a caching simulator that takes transaction traces as input and outputs hit rates for the offline Belady and Transactional Belady algorithms.

### Scope

The artifact enables others to run DeToX directly. All code used in the paper is made available.

### Contents

The artifact consists of a Github repository hosted at <https://github.com/audreycccheng/detox>. The repository is structured as follows:

- /chronocache - the codebase for ChronoCache
- /oltpbench-chronocache - the benchmarks for ChronoCache
- /redis - the modified version of Redis supporting transactional caching algorithms
- /simulator - the caching simulator for offline policies
- /sys - the transactional caching system
  - /benchmarks - the benchmarks for running DeToX
  - /src - the implementation of the DeToX shim layer

### Hosting

The artifact is hosted at <https://github.com/audreycccheng/detox> on the main branch at commit 604c9bd.

### Requirements

The following packages are required to run the codebase.

- mvn 3.8.5
- build-essential
- Java 17

For specific installation guides for each system, please see the Github repository.



# Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction

Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu\*

*Tsinghua University*

## Abstract

Combining persistent memory (PM) with RDMA is a promising approach to performant replicated distributed key-value stores (KVSs). However, existing replication approaches do not work well when applied to PM KVSs: 1) Using RPC induces software queuing and execution at backups, increasing request latency; 2) Using one-sided RDMA WRITE causes many streams of small PM writes, leading to severe device-level write amplification (DLWA) on PM.

In this paper, we propose ROWAN, an efficient RDMA abstraction to handle replication writes in PM KVSs; it aggregates concurrent remote writes from different servers, and lands these writes to PM in a sequential (thus low DLWA) and one-sided (thus low latency) manner. We realize ROWAN with off-the-shelf RDMA NICs. Further, we build ROWAN-KV, a log-structured PM KVS using ROWAN for replication. Evaluation shows that under write-intensive workloads, compared with PM KVSs using RPC and RDMA WRITE for replication, ROWAN-KV boosts throughput by 1.22× and 1.39× as well as lowers median PUT latency by 1.77× and 2.11×, respectively, while largely eliminating DLWA.

## 1 Introduction

Replicated distributed key-value stores (KVSs) support many applications by providing durability and high availability [28, 56, 76]. The recent commercialization of persistent memory (PM), e.g., Intel’s Optane DIMMs, enables local storage with extremely low latency (e.g., ~100ns when persisting small data [73]). When building replicated distributed KVSs with such fast storage media, network and CPU will become determinants of request latency, since replicating an object (i.e., key-value pair) involves multiple times of network communication and request queuing/execution.

RDMA, a widely-deployed network technology [34, 37, 53], is promising to mitigate the network and CPU overhead. First, RDMA delivers low latency (~2μs) due to protocol-offload RDMA NICs (RNICs) and kernel-bypass software. Second, RDMA provides one-sided WRITE and READ, allowing remote memory accesses without involvement of remote CPUs. Recent work have leveraged WRITE to replicate data in DRAM (i.e., WRITE-enabled replication) [17, 30, 31, 69]. This eliminates software queuing/execution of backups in the critical

path, thus significantly cutting the replication latency compared with RPC-enabled replication.

Yet, in the context of PM KVSs, WRITE-enabled replication approach does not work well: it induces severe device-level write amplification (DLWA) on PM. Specifically, a KVS is typically finely sharded for load balancing and fast recovery, so every server acts as backups for many shards, receiving numerous concurrent replication writes from many remote threads; besides, these replication writes are typically small (~100B) due to prevalent tiny objects in real-world workloads [24, 52]. In WRITE-enabled replication approaches (e.g., FaRM [31]), each server allocates an exclusive backup log for *every remote thread*, to accommodate remote WRITE from primaries. When adopting WRITE-enabled replication to PM KVSs, these backup logs generate a huge number of PM write streams<sup>1</sup>, which contain lots of small-sized writes. These numerous write streams lead to severe DLWA, since PM has block access granularity at media level (e.g., 256B in Optane DIMMs) and its hardware combining capacity is bounded. In our experiments, with 128B RDMA WRITE, 144 remote PM write streams cause 1.58× DLWA (§2.4). DLWA wastes limited PM write bandwidth, shortens PM lifetime, and harms PM’s persistence efficiency.

In this paper, we propose ROWAN, an efficient RDMA abstraction to handle replication writes on PM KVSs. ROWAN can aggregate numerous concurrent remote writes from different servers, and land these writes to PM *sequentially*, so as to largely eliminate DLWA. Besides, it is one-sided as RDMA WRITE, enabling backup-passive replication with low latency and high CPU efficiency. We realize ROWAN with off-the-shelf RNICs based on two observations: 1) RDMA SEND is two-sided on the control path but *one-sided on the data path*; 2) RNICs consume receive buffers *in order*. Thus, we let a control thread at the receiver side push PM-resident buffers into receive queues *in increasing address order*. Senders only need to issue SEND for remote PM writes and wait for ACKs generated by receiver-side RNICs. We leverage two RNIC hardware features, shared receive queue (SRQ) [11] and multi-packet receive queue (MP RQ) [7, 9], to merge writes from different connections and support variable-sized writes, respectively. We also streamline ROWAN’s control path by minimizing the control thread’s tasks. A ROWAN instance can

\*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn)

<sup>1</sup>A *write stream* is a group of writes targeting contiguous addresses, e.g., writes that perform log appending.



achieve 54.5Mops/s for highly concurrent 64B remote PM writes, with almost no DLWA.

Further, we build ROWAN-KV, a PM KVS leveraging ROWAN for primary-backup replication. It adopts a log-structured approach to manage both local PM writes and remote PM writes. Specifically, each server maintains per-thread primary logs and a *single* backup log on PM. For a PUT request, a worker thread in servers generates a log entry containing the targeted object; then, it persists the log entry into its local primary log via CPU instructions and every backup’s backup log via one-sided ROWAN. For a GET request, the thread searches DRAM-resident indexes which point to objects in logs. In this way, ROWAN-KV features high performance and low DLWA: 1) Replication bypasses CPUs of backups, ensuring low latency and saving CPU cycles for foreground operations; 2) The number of PM write streams in a server is small (i.e.,  $n$  primary logs + 1 backup log, where  $n$  is local thread count), enabling efficient write combining in PM hardware and thus largely eliminating DLWA. ROWAN-KV also introduces a failover mechanism for fault tolerance and a dynamic resharding mechanism for load balancing.

We evaluate ROWAN-KV on Optane DIMMs under a cluster of 14 machines (8 clients and 6 servers). Our evaluation focuses on YCSB benchmarks [26] with object sizes from three typical Facebook KVSs workloads [24] (i.e., ZippyDB, UP2X and UDB). Compared with KVSs using RPC and WRITE for replication, ROWAN-KV boosts throughput by 1.22× and 1.39×, lowers median PUT latency by 1.77× and 2.11×, and lowers 99% latency by 1.26× and 2.06×, respectively, under write-intensive workloads. In addition, the DLWA is less than 1.032× in ROWAN-KV, while 1.54× in the WRITE-enabled KVSs. Under read-intensive workloads, they have similar performance. We also compare ROWAN-KV with two software techniques mitigating DLWA, i.e., batching and log sharing; ROWAN-KV still outperforms them.

In summary, this paper makes the following contributions:

- It demonstrates that WRITE-enabled replication can lead to severe device-level write amplification on PM KVSs.
- It introduces ROWAN abstraction and ROWAN-KV with goals of low latency and low device-level write amplification.
- It uses experiments to confirm the efficacy of ROWAN-KV.

## 2 Background and Motivation

In this section, we first provide the background on PM (§2.1) and RDMA (§2.2). Then, we show that characteristics of typical KVSs architecture and workloads together lead to high fan-in small writes for replication (§2.3). Finally, with experiments, we demonstrate that when handling these writes, WRITE-enabled replication causes severe DLWA (§2.4).

### 2.1 Persistent Memory (PM)

PM is a new kind of storage device that sits on the memory bus. Thus, PM is byte-addressable and can be accessed by CPUs via load/store instructions. In this paper, we focus on Intel’s Optane DIMM, the only available PM product.

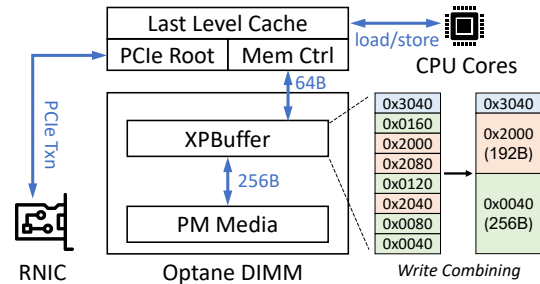


Figure 1: Architecture of Optane DIMMs and RNICs.

**PM performance.** Optane DIMMs have unique performance characteristics. In terms of bandwidth, an Optane DIMM offers about 2GB/s for writes and 6GB/s for reads, which are 1/6 and 1/3 of DRAM, respectively. In terms of latency, compared to DRAM, Optane DIMMs have similar write latency but 3× higher read latency [73]. The limited write bandwidth and high read latency of Optane DIMMs are the main design considerations for many PM systems [20, 25, 48, 49, 59, 67, 74].

**PM architecture.** Figure 1 presents the architecture of Optane DIMMs. The memory controller generates cache-line granularity (i.e., 64B) read/write requests to Optane DIMMs, but the internal PM media has a 256B access granularity (referred as *XPLine* in this paper). Such a granularity mismatch will trigger read-modify-write events, thus leading to device-level write amplification (DLWA). To mitigate DLWA, each Optane DIMM features an XPBuffer [73], which performs write combining for adjacent 64B writes, as shown in the right part of Figure 1. Yang et al. estimated that the XPBuffer in an Optane DIMM is approximately 16KB in size [73].

**Persistent modes.** There are two persistent modes for PM: ADR and eADR [36]. In ADR mode, once a store reaches the memory controller, it can survive power failure; but the CPU cache is volatile, so programmers must explicitly flush data from the CPU cache (using `clwb` or `clflushopt` instructions) or bypass the CPU cache (using `ntstore` instructions). In eADR mode, the CPU cache also belongs to the persistence domain: its data will be flushed to PM upon power failure.

### 2.2 Remote Direct Memory Access (RDMA)

RDMA is a network technology that offers high bandwidth (e.g., 100 Gbps) and low latency ( $\sim 2\mu\text{s}$ ).

**Verb types.** RDMA provides two types of verbs for network communication: *message verbs* and *memory verbs*. Message verbs, i.e., SEND and RECV, are the same as Linux socket interfaces: a SEND emits a message to a remote server that prepares receive buffers via RECV. Memory verbs include WRITE, READ and ATOMIC. These verbs can operate receivers’ memory without involving receivers’ CPUs. Due to the *one-sided* feature, memory verbs enjoy low latency and high CPU efficiency.

**Queue pair.** RDMA servers use queue pairs (QPs) for communication. A QP contains a *send queue* (SQ) and a *receive queue* (RQ). A server posts requests, including SEND, WRITE, READ, and ATOMIC, to the send queue, and posts RECV to the receive queue for accommodating incoming SEND messages. A send/receive queue is associated with a *completion queue*

	max shard size	# of backup shards (stored by one PM server)
CosmosDB	20GB [10]	200
DynamoDB	10GB [2]	400
FoundationDB	500MB [3]	8,400
Cassandra	100MB [1]	42,000
TiKV	96MB [38]	43,000

**Table 1:** A PM server hosts many backup shards for popular KVSs. We assume 3-way replication and a typical configuration of PM servers: 2 sockets, each with 3TB Optane DIMMs (6TB in total).

(CQ), which generates completion signals for posted verbs.

**Remote persistence.** When issuing a WRITE to remote PM, to ensure the data persistence, we should take two extra actions. ① Since receiver-side RNICs return acknowledgements before data in WRITE is DMA-ed to PM, we should send a READ (1B in arbitrary addresses) to flush RNIC and PCIe buffers at the receiver side [42]. These two verbs (i.e., WRITE followed by READ) can be posted in one request according to the ordering guarantee of RDMA [70]. ② We should disable *Data Direct I/O (DDIO)* [5, 32], a technology of Intel CPUs that lets RNICs directly DMA data to last level cache (LLC). In ADR mode, disabling DDIO ensures that DMA-ed data can reach persistence domain. In eADR mode, it avoids PM write amplification resulting from LLC’s near-random eviction (64B cache line vs. 256B XPLine) [42, 70].

### 2.3 High Fan-in Small Writes in KVSs

In KVSs, replication makes *high fan-in small writes* a dominant access pattern due to the following two reasons.

**1) Data sharding.** Distributed storage systems (including KVSs) typically split the entire data set into a large number of shards, and then distribute these shards across many servers [16, 50]. Each shard has multiple replicas, with one selected as *primary* and the others as *backups*. Data sharding has two advantages. First, it can improve load balancing and support dynamic data migration in a fine-grained manner. Second, it can improve availability: when a server fails, since replicas of its data are distributed to many servers, the system can perform recovery and re-replication in parallel. For example, FaRM [30] maps each server into 100 consistent hashing rings by default; in Facebook’s RocksDB clusters, each server typically hosts tens or hundreds of shards [29].

With data sharding, each server acts as backups for tens or hundreds of shards, and their primaries are distributed to many servers. This makes every server receive messages for data replication, i.e., replication writes, from many primaries residing in many other servers. We call it *high fan-in writes*.

To solidify the argument of high fan-in writes in KVSs, we analyze five widely-used replicated KVSs. As shown in Table 1, these KVSs all have a maximum shard size, from tens of megabytes (i.e., Cassandra [1] and TiKV [38]) to several gigabytes (i.e., DynamoDB [2] and CosmosDB [10]). When we deploy these KVSs on servers having terabytes of PM, each server will host a considerable number of backup shards which ranges from 200 (CosmosDB) to 43,000 (TiKV),

generating high fan-in replication writes.

The degree of fan-in is even higher in systems equipped with fast network hardware (e.g., RNIC) [30, 31, 44, 45, 61, 69]. To achieve multicore-scalable and squeeze out the raw performance of NICs, these systems run multiple threads, each independently processing requests using exclusive network connections. For example, in DrTM+H [69], every worker thread independently issues RDMA WRITE for replication. With this threading model, the degree of fan-in increases from the number of remote servers to the *number of remote threads*.

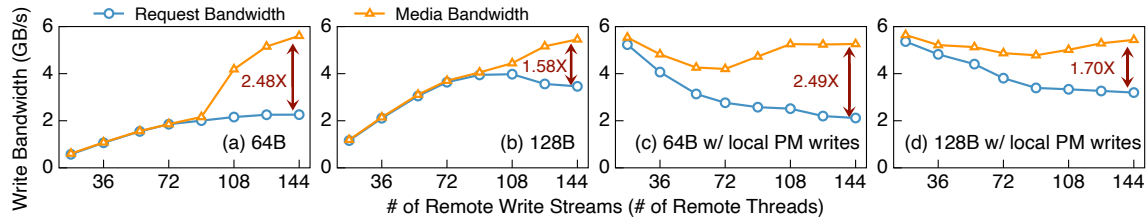
**2) Numerous small-sized objects.** Many important applications relying on KVSs generate numerous small objects, whose size is much smaller than the access granularity of PM media (e.g., 256B XPLine in Optane DIMMs). For example, in ZippyDB, the largest KVS at Facebook [15], the average size of objects is only 90.8B [24]. Moreover, the other two typical KVSs at Facebook — UP2X (a KVS for AI services) and UDB (a KVS for social graph) — have average object size of 57.25B and 153.8B, respectively [24]. Twitter exhibits a similar workload feature: the most common length of a tweet is only 33 characters [14, 52]. This paper focuses on these small objects because of their prevalence and importance.

When a KVS handles PUT requests (from clients) for these small objects, primaries emit replication writes to associated backups. These writes are small, since they typically only contain replicated objects with tiny metadata [56]. These writes are also high fan-in due to data sharding, as explained before. As a result, we can conclude that *high fan-in small writes are a dominant access pattern in the cluster of KVSs*.

### 2.4 DLWA from WRITE-enabled Replication

Recent research demonstrates that for in-memory DRAM systems, compared with RPCs, leveraging RDMA WRITE for replication can obtain significant performance gain [17, 30, 31, 69]. In such WRITE-enabled replication, primaries issue replication writes to backups’ logs via one-sided WRITE, and only need to wait for acknowledgements (ACKs) from the RNIC hardware of backups. This eliminates software queueing/execution of backups in the critical path, thus enjoying low latency (e.g., Mu [17] cuts the latency by 61%). Further, the saved CPU cycles in backups can serve requests (e.g., GET) from clients, thus improving system throughput.

In systems using WRITE-enabled replication, to handle high fan-in replication writes from many remote threads (recall §2.3), each server maintains lots of backup logs, each accommodating WRITE from an individual remote thread (which can act as primary) [31, 69]. For example, in FaRM’s evaluation with 90 machines (each running 30 worker threads) [31], there are thousands of backup logs (i.e., 89×30) in each server. Yet, when we apply WRITE-enabled replication to PM KVSs, these backup logs (which are placed in PM for durability) will cause a huge number of PM write streams, which contain lots of small writes, thus inducing severe DLWA. We conduct an experiment to demonstrate it.



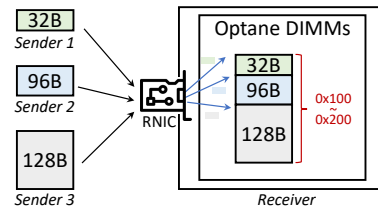
**Figure 2:** DLWA with varying remote write streams.  $DLWA = \text{media bandwidth}/\text{request bandwidth}$ . Threads access PM on a remote server; each thread generates a remote write stream. In (c) and (d), 18 CPU cores in the remote server perform local sequential PM writes.

In the experiment, we launch a number of threads (on four servers), each issuing sequential RDMA WRITE to an exclusive PM-resident log in a remote server and thus generating a PM write stream. We disable DDIO in the remote server and post a READ after each WRITE to guarantee persistence. The remote server is equipped with three 256GB Optane DIMMs and a 100Gbps RNIC. We use `ipmctl` [8] to periodically read hardware counters of Optane DIMMs, calculating *request bandwidth* and *media bandwidth*, which means write bandwidth received from memory bus and write bandwidth issued to PM media, respectively. Figure 2(a) and (b) show results with 64B and 128B WRITE size (representing small replication writes, §2.3), respectively. When remote write stream count is lower than 90, DLWA is negligible. This is because the XPBuffer on Optane DIMMs can combine adjacent small writes from the same write streams into 256B internal writes (§2.1). However, the capacity of combining is bounded due to the limited size of XPBuffer. Consequently, as the number of remote write streams continues to increase, severe DLWA appears. Specifically, when remote write stream count is 144, the DLWA is 2.48 $\times$  and 1.58 $\times$  in case of 64B WRITE and 128B WRITE, respectively.

Next, we consider a more practical scenario where local PM writes exist. In the remote server, we run 18 CPU cores, each performing sequential 128B PM writes using `ntstore`. We repeat the above experiment; Figure 2(c) and (d) show the results. Without remote RDMA WRITE, local PM writes can deliver high request bandwidth (i.e., available bandwidth). As the remote write stream count increases, DLWA in Optane DIMMs reaches 2.49 $\times$  and 1.70 $\times$  in case of 64B WRITE and 128B WRITE, respectively. In addition, the available bandwidth drops from 5.2GB/s to 2.1GB/s (60%) for 64B WRITE, and from 5.4GB/s to 3.2GB/s (41%) for 128B WRITE.

DLWA on PM leads to three issues. First, it reduces available PM write bandwidth, thus degrading system performance. The wasted bandwidth could also have been used for co-located applications [33, 54, 55]. Second, it shortens the lifetime of PM which has limited write endurance [6]. Third, severe DLWA consumes a considerable number of hardware resources (e.g., XPBuffer), harming persistence efficiency.

To efficiently handle high fan-in small writes, we need a new RDMA abstraction (rather than WRITE) for PM KVSS. This abstraction should *mitigate DLWA, while achieving benefits of one-sided verbs — low latency and high CPU efficiency.*



**Figure 3:** An instance of RowAN abstraction.

### 3 RowAN Abstraction

We propose RowAN, a new RDMA abstraction to handle high fan-in small writes in PM KVSSs. In this section, we first describe RowAN’s semantic and characteristics. Then, we present how to realize RowAN using off-the-shelf RNICs.

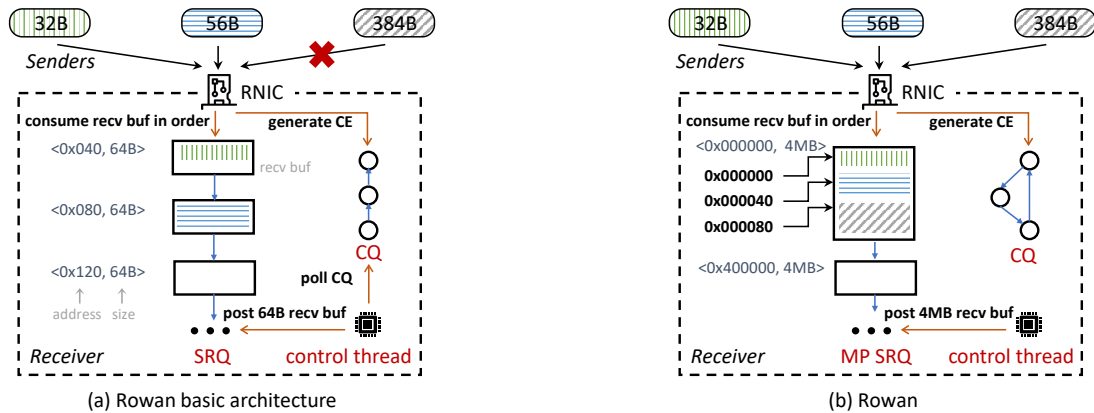
#### 3.1 RowAN Semantic

Figure 3 presents a RowAN instance. A RowAN instance is associated with one receiver and a set of senders. Senders concurrently issue writes to the receiver which has registered a large PM area. The receiver-side RNIC lands these writes to the PM area *sequentially*, and finally returns ACKs to senders.

RowAN abstraction has the following advantages. First, by translating concurrent remote small writes into a single write stream, the XPBuffer in Optane DIMMs can easily combine them into 256B XPLine writes, largely eliminating DLWA. Second, since all the data operations are performed by the receiver-side RNIC without involving receiver-side CPUs, RowAN enjoys benefits of low latency and high CPU efficiency like RDMA WRITE. In addition, compared with CPUs, RNIC ASICs can deliver extremely high throughput.

**Comparison with batching.** Batching is also an approach that can mitigate DLWA on PM: it opportunistically accumulates multiple small writes at the sender side, and then emits the batched writes to the receiver via one RDMA WRITE. However, batching induces extra latency, sapping the benefits of extremely low-latency hardware (i.e., RNICs and PM). In contrast, RowAN does not delay any write and thus ensures low latency: senders *immediately* issue writes and receiver-side RNICs *immediately* land received writes to PM. In addition, as we will show in §6, batching frequently fails to accumulate enough small writes within a short time interval in KVSSs, and RowAN outperforms batching in both latency and throughput. Our view of batching has been echoed by authors of RAMCloud — “. . . batching requires some operations to be delayed until a full batch has been collected, and **this is not acceptable in a low-latency system such as RAMCloud**” [56].





**Figure 4:** Realizing ROWAN with off-the-shelf RNICs. (a) ROWAN basic architecture using shared receive queue (SRQ). In this subfigure, the 32B write and 56B write are placed in the same XPLine. Yet, the 384B write fails to be received due to 64B receive buffers. (b) ROWAN using multi-packet shared receive queue (MP SRQ) with 64B stride. In this subfigure, three writes are placed in two XPLines of the first receive buffer. We use a completion queue (CQ) ring to eliminate CQ polling in the control thread.

## 3.2 High-Performance ROWAN

ROWAN is conceptually simple but challenging to realize using off-the-shelf RNICs. We do not want to modify RNIC hardware like StRoM [60] and PRISM [23], so as to enable ROWAN to be deployed immediately in datacenters today that are equipped with RNICs. Before describing our solution, we present a straightforward solution that has poor performance.

### 3.2.1 Straightforward Solution

A straightforward solution to realize ROWAN abstraction is combining RDMA WRITE and atomic verb FETCH\_AND\_ADD. Specifically, there is a 64-bit sequencer stored in the receiver’s memory. When performing a write, the sender first issues a FETCH\_AND\_ADD to the sequencer, reserving a PM address; then, it issues a WRITE to this address. This solution has two limitations. First, it needs two round trips, increasing the latency. Second, the poor performance of atomic verbs bottlenecks throughput: even storing the sequencer in RNICs’ device memory [68], the throughput is less than 10Mops/s.

### 3.2.2 Our Solution

Counter-intuitively, we use RDMA SEND and RECV to realize ROWAN. This is based on our two observations.

- **RDMA SEND is two-sided on the control path but one-sided on the data path.** In the control path, the receiver’s CPUs prepare receive buffers via RECV; however, in the data path, when handling SEND requests, the receiver-side RNIC performs *all* tasks, including landing SEND’s data to receive buffers and returning ACKs.
- **In a receive queue, receive buffers are consumed in order.** Every time, the receiver-side RNIC pops the *first* buffer in the associated receive queue and lands data to it.

**Key idea.** On the control path, CPUs push PM buffers into the receive queue in increasing address order; on the data path, the receiver-side RNIC consumes them in order.

**Basic architecture.** Figure 4(a) shows the basic architecture of ROWAN implementation. ROWAN uses reliable connection (RC) QPs to delegate transmission reliability to RNICs. We create a shared receive queue (SRQ) [11] which is associated

with all QPs; thus, RNICs can land data of SEND from *different remote QPs* to the same receive queue. In the receiver, we reserve a dedicated thread, namely *control thread*, to perform control-path tasks; the RNIC performs data-path tasks.

Specifically, the control thread splits the PM area into fixed-sized (e.g., 64B in Figure 4(a)) buffers, and posts these buffers (using RECV) into the SRQ in *increasing address order*. Senders encapsulate writes into SEND requests, and emit them to the receiver; each SEND is followed by a READ for persistence. When receiving a SEND (followed by a READ), the receiver-side RNIC pops the first buffer in SRQ, DMA’s the SEND’s data into the buffer, generates a completion entry (CE) to the SRQ’s CQ, and finally returns an ACK to the sender. In this way, writes from different senders can be combined into the same XPLines on PM, mitigating DLWA.

**Handling variable-sized writes.** When the size of a SEND’s data is larger than the first buffer in the SRQ, the RNIC cannot accommodate it and will trigger an error CE. For example, in Figure 4(a), with 64B receive buffers, the 384B write cannot be handled. If we use a buffer size larger than 256B for the SRQ to support relatively large writes, small writes from different senders will not be combined into the same XPLines, destroying the benefits of ROWAN abstraction.

Fortunately, current RNICs (e.g., ConnectX-4/5/6) support a new type of RQ, called *multi-packet receive queue* [7, 9] (MP RQ). In an MP RQ, each receive buffer can accommodate *multiple* SEND requests. We need to define a *stride* (e.g., 64B) for an MP RQ. When receiving a SEND, the RNIC appends the data to the receive buffer that is being used, and the start address is stride-aligned. If there is no enough space left, the RNIC pops a new receive buffer from the MP RQ to use.

Figure 4 shows ROWAN that uses MP SRQ, where we set the stride to 64B and receive buffer size to 4MB. In the figure, three writes are placed in two XPLines (i.e., 512B area) in the first receive buffer, each having a 64B-aligned start address. By using MP SRQ, ROWAN can support variable-sized writes, while combining small writes to mitigate DLWA.



There are two points worth noting when using MP SRQ:

- In ROWAN, the stride is a fixed value of 64B. We do not choose a smaller value (e.g., 32B) for two reasons. First, in the RNIC we use (i.e., ConnectX-5), the minimum supported stride value is 64B. Second, recent studies suggest that senders should pad small writes to PCIe data word (64B) granularity [70], to avoid expensive read-modify-write operations on receivers' PM. Thus, we assume the incoming small writes are already 64B granularity.
- If a SEND is larger than maximum transmission unit (MTU), it is comprised of multiple packets. The RNIC may land these packets to non-contiguous addresses. We let the upper applications (e.g., KVSs) to handle this case.

**Minimizing control-path tasks.** On ROWAN's data path, the receiver-side RNIC can deliver extremely high throughput (> 50Mops/s). On the control path, for CPU efficiency, we only want to use *one* control thread; thus, we minimize control-path tasks to make them can be easily handled by one thread.

There are two tasks performed by the control thread: posting receive buffers into the MP SRQ and polling the CQ to consume CEs. For the former, since we use large receive buffers (e.g., 4MB) by leveraging the multi-packet feature and post a batch of receive buffers at a time, this task is lightweight. For the latter, unfortunately, unlike other verbs, RECV can not be marked as un signaled, so every SEND will generate a CE at the receiver side. The control thread cannot timely consume these CEs (considering > 50Mops/s throughput), making the CQ fill and thus causing QPs in an error state. We get inspiration from eRPC [43] to address this problem. Like eRPC, we create a CQ that forms a *ring structure*, so that the RNIC can overwrite entries in the CQ ring in a round-robin manner. In this way, the control thread does not need to poll the CQ.

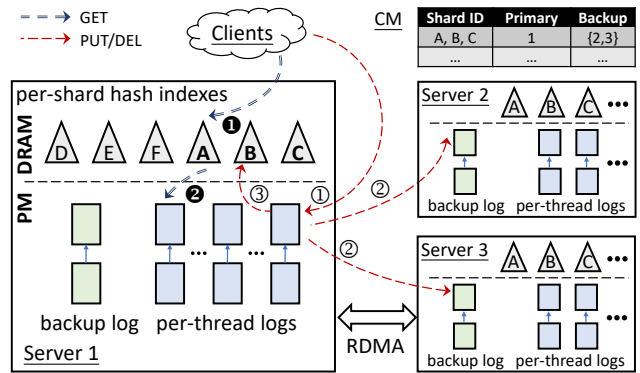
## 4 ROWAN-KV Design

We build ROWAN-KV, a PM KVS that uses ROWAN for primary-backup replication. It has two main design goals.

- **Low latency.** ROWAN-KV exploits one-sided ROWAN to eliminate software overhead at backups during replication.
- **Low DLWA.** ROWAN-KV adopts a log-structured approach to manage PM writes from both local CPUs and remote CPUs. For the former, every thread appends data in its local log. For the latter, ROWAN merges replication writes into a *single* backup log. Hence, Optane DIMMs only receive a small number of write streams and can efficiently combine adjacent small writes into XPLines, thus mitigating DLWA.

### 4.1 Overview

Figure 5 shows the architecture of ROWAN-KV. Servers persistently store objects (i.e., key-value pairs) in PM and use RDMA for network communication. ROWAN-KV divides the entire data set into many shards and distributes them across servers. Each shard is replicated for high availability: with the replication factor of  $k$ , it has one server as *primary* and  $k-1$  servers as *backups*. Clients issue KV requests via RPCs. **Sharding mechanism.** ROWAN-KV hashes each object's key



**Figure 5:** Architecture of ROWAN-KV. The *per-thread logs (t-logs)* and *backup log (b-log)* are divided into 4MB segments.

into a 64-bit number and lets a shard manage a continuous range in the hashed keyspace. Shard distribution is maintained by a *configuration manager (CM)* and is cached in servers and clients. ROWAN-KV uses a dynamic resharding mechanism to mitigate load imbalancing from overloaded servers (§4.6).

**Log-structured approach.** ROWAN-KV adopts a log-structured approach, where each server has three components:

- **Per-thread logs.** Each server launches a number of *worker threads* to handle requests from clients. Each worker thread maintains a per-thread log (**t-log**) in PM, which stores objects of PUT/DEL requests. We do not allocate independent logs for each shard, to reduce random PM writes.
- **Backup log.** Each server has a *single* backup log (**b-log**) in PM, which receives replication writes from primaries using a ROWAN instance. By doing so, ROWAN-KV can largely eliminate DLWA from high fan-in small writes.
- **Per-shard hash indexes.** Each server builds a DRAM-resident hash table for every shard it manages, to index objects in t-logs or the b-log. Putting indexes in DRAM can avoid random PM writes and expensive PM reads [22, 25]. The t-logs and b-log are divided into 4MB *segments*.

**Handling KV requests.** When issuing a KV request for an object, the client sends an RPC to a worker thread residing in the server that is the targeted shard's primary.

For a PUT/DEL request, the worker thread generates a log entry containing the object (only the object's key for DEL), and persistently appends the log entry to its local t-log using `ntstore` instructions (① in Figure 5). Then, the worker thread issues replication write for every backup via one-sided ROWAN, persistently appending the log entry to every backup's b-log (②). Upon receiving all ACKs from backups' RNICs, the worker thread updates the associated index to make the object (in t-logs) visible (③), and finally returns a response to the client. ROWAN-KV has a strong durability guarantee: when a client receives the response of a PUT/DEL request, its effects have been persisted on all replicas.

For a GET request, the worker thread first locates the object by searching the associated index (①). Then, it copies the object's value from t-logs (②) and replies to the client.

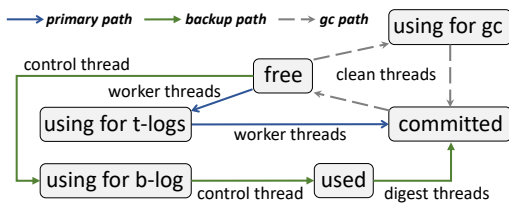


Figure 6: Life cycle of segments.

**Background operations.** ROWAN-KV uses three types of threads to perform background operations.

- *Control thread.* One control thread performs control-path tasks for the ROWAN instance (§3). In ROWAN-KV, it pushes free segments to the b-log via RDMA RECV, and hands used segments over to digest threads.
- *Digest threads.* There are multiple digest threads. They digest used segments from the b-log. Specifically, they parse log entries and update associated indexes.
- *Clean threads.* There are multiple clean threads. They garbage collect stale objects in segments (from worker threads or digest threads) to reclaim free PM space.

## 4.2 Log Metadata

In ROWAN-KV, t-logs and b-log are comprised of multiple segments, each storing a number of log entries. We describe segment metadata and log entry metadata, respectively.

### 4.2.1 Segment Metadata

A segment’s metadata mainly includes its *state*. At any given time, each segment is in one of four states:

- *Free.* The segment can be allocated to t-logs by worker threads, the b-log by the control thread, or clean threads.
- *Using.* The segment is being used by t-logs, the b-log, or clean threads; it has space to store new log entries.
- *Used.* It has no space to store new log entries, and some of its log entries have *not* been persisted on all replicas.
- *Committed.* It has no space to store new log entries, and all of its log entries have been persisted on all replicas.

In addition to the state, a segment has an extra metadata called *owner*, indicating which type of thread allocates it (e.g., worker threads). Each server maintains a PM array called *segment meta table* to record metadata for all its segments.

Figure 6 presents the life cycle of segments. The path for primaries is simple: a worker thread allocates a *free* segment for its t-log, and the segment becomes *using* state. Once the segment has no space, it transitions into *committed*, since the worker thread can easily ensure that all of the segment’s log entries have been persisted on all replicas. The path for backups is fairly complicated, where we should accurately distinguish between *used* segments and *committed* segments (§4.3 and §4.4). Such a distinguishment is essential for failover (§4.5).

### 4.2.2 Log Entry Metadata

A log entry contains the request type (i.e., PUT/DEL) and the targeted object (only the object’s key for DEL). It also includes three metadata fields:

- *32-bit checksum.* The checksum covers the whole log entry. Checksums eliminate persistent tails for logs: upon

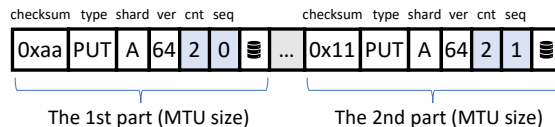


Figure 7: A 2-MTU-sized log entry in the b-log.

recovery, we can identify the end of each log by calculating checksums. Besides, backups can use checksums to independently check the integrity of log entries in the b-log.

- *48-bit version.* Each shard has a version, namely *shard version*, which is maintained by its primary. Upon a PUT/DEL request, the worker thread atomically increments the associated shard version, and stores the obtained version into the log entry. Upon recovery, the version allows us to identify the most recent objects from multiple t-logs.
- *16-bit shard ID.* It indicates which shard the targeted object belongs to.

**Handling larger-than-MTU log entries.** For a log entry that is larger than MTU, backup-side RNICs may divide it into multiple packets and place them in non-contiguous addresses of the b-log (recall §3.2). To enable backups check the integrity of such a log entry, we design a simple counter-based metadata. Specifically, if a log entry is larger than MTU, we logically divide it into multiple MTU-sized blocks, and duplicate log entry metadata at the start of each block (each *checksum* field protects the individual block). Besides, we add two extra metadata to each block: 1) *cnt*: block count of the log entry, and 2) *seq*: the sequence number of the block.

Figure 7 shows a 2-MTU-sized log entry in the b-log, where its two blocks are not adjacent. The pair of  $\langle \text{shard ID} : A, \text{version} : 64 \rangle$  uniquely identifies the log entry. When scanning the two blocks (checksums match) with their *cnt* and *seq*, backups can determine the log entry’s integrity.

## 4.3 Managing the Backup Log

The control thread manages the b-log by communicating with the RNIC and digest threads. To minimize the communication overhead, the control thread performs tasks in a *batch* manner.

Specifically, when the system starts up, the control thread allocates a considerable number of free segments (e.g., 512) for the b-log, and pushes them into ROWAN’s MP SRQ via RECV. Then, it enters into a loop: 1) identifies a batch of segments (e.g., 128) that is in *used* state; 2) hands these segments over to digest threads; 3) allocates a batch of free segments and pushes them into the b-log via one RECV call. Note that a free segment transitions into the *using* state after it is allocated by the control thread (recall backup path in Figure 6).

**Identifying used segments.** The control thread adopts a simple method to identify used segments in the b-log. For every segment pushed into the b-log, its first 64 bits are set to zeros. Meanwhile, the first 64 bits in a log entry include the request type, which is non-zero. Thus, when the control thread finds that a segment has non-zero first 64 bits, it can ensure that all *previous* segments in the b-log (we call the set of segments *S* here) have been allocated by the RNIC for accommodating

log entries. However, this does not mean that segments in  $S$  are *used*, since maybe some DMA operations writing log entries in  $S$  are outstanding. Hence, we wait 2ms for all these DMA operations to complete, to guarantee that all segments in  $S$  have transitioned into *used* state. At the primary side, worker threads measure the time of each replication write: if it is more than 1ms, worker threads retry the replication write.

#### 4.4 Digest and Garbage Collection

**Digest.** Multiple digest threads process used segments in the b-log in parallel. Each digest thread manages an exclusive set of shards: it extracts log entries from used segments in order and only processes shards it manages. For a log entry, digest threads update the index of the associated shard. Besides, digest threads identify *committed* segments, and hand these segments over to clean threads.

**Identifying committed segments.** To help digest threads identify committed segments in the b-log, primaries disseminate the information of log entries to backups. Specifically, for a shard, worker threads in its primary maintain a `CommitVer`; any log entry containing a version  $\leq$  `CommitVer` has been persisted on all replicas. Every 15ms, worker threads write the  $\langle$ shard ID, `CommitVer` $\rangle$  pair into backups' b-logs via `ROWAN`.

At the backup side, digest threads maintain an array `CommitVerArray`, which contains associated `CommitVer` for each shard. When encountering a  $\langle$ shard ID, `CommitVer` $\rangle$  during parsing segments of the b-log, digest threads update `CommitVerArray`. Meanwhile, when processing a segment, digest threads generate an array `MaxVerArray` for it; for each shard, this array records the *maximum version* that digest threads have encountered in log entries. A used segment can transition into committed one, if its `MaxVerArray`  $\leq$  `CommitVerArray` (i.e., for every shard, the maximum version in `MaxVerArray`  $\leq$  `CommitVer` in `CommitVerArray`).

**Garbage collection.** Multiple clean threads garbage collect stale objects in committed segments. When memory utilization of a committed segment, i.e., the percentage of valid bytes, is lower than a pre-defined threshold (e.g., 75% in our evaluation), a clean thread cleans it. Specifically, the clean thread scans the committed segment and checks the liveness of objects in log entries (by searching indexes). For live objects, the clean thread copies associated log entries to a *using* segment and updates indexes. Finally, the committed segment transitions into *free* state for future usages.

#### 4.5 Failover

We adopt FaRM's reconfiguration-style approach [31] to handle failover but tailor it for `ROWAN-KV`. A *configuration* in `ROWAN-KV` contains 1) 64-bit term, 2) membership, i.e., the set of live servers, and 3) shard distribution. The configuration is persistently stored in a Zookeeper instance [40], and is cached in the CM, clients, and servers. `ROWAN-KV` uses leases to detect failures for servers and CM [31]. When the CM fails, `ROWAN-KV` activates a new CM using the same mechanism as FaRM [31]. When a server fails, `ROWAN-KV` performs failover

with the following three phases.

**1) Generating and committing a new configuration.** The CM generates a new configuration, where the term is incremented and the membership excludes the failed server. In the new shard distribution, the CM reassigns shards managed by the failed server to live servers, and promotes a backup to the new primary for each shard losing its primary.

Then, the CM stores the new configuration in Zookeeper and sends it to all servers. Servers cache the configuration, destroy QPs used for communicating with the failed server, and respond. From this point, servers block all requests from clients. Once the CM receives all responses, after ensuring that the lease for the failed server has expired, it sends a commit message to all servers. Now, servers can unblock requests. A server rejects requests containing terms that are lower than the one it caches. Clients will fetch the new configuration from CM upon receiving rejected responses.

**2) Promoting backup to primary.** When a backup of a shard (we call the shard  $A$  here) is promoted to the new primary, its worker threads block requests to  $A$  until digest threads build indexes for all objects of  $A$ . The new primary and backups should reach a consensus on the committed log entries. Hence, the new primary and backups process *using* and *used* segments in the b-log, collecting log entries belonging to  $A$ . These collected log entries are gathered to the new primary and then are scattered to backups. The new primary and backups store these log entries into segments. In this way, all replicas will own the same set of log entries for  $A$ . During digest, the new primary constructs a valid shard version for  $A$ , which is larger than versions in any  $A$ 's log entry.

**3) Re-replication.** The CM adds a new backup for the shard having replicas in the failed server. The new backup performs re-replication asynchronously. It first initializes an index for the shard, and then sends a message to the primary. Upon receiving the message, the primary traverses the shard's index and transmits associated log entries to the new backup.

#### 4.6 Dynamic Resharding

`ROWAN-KV` introduces a dynamic resharding mechanism to migrate hotspot shards for improving load balancing.

CM detects overloaded servers and produces new shard distribution. Specifically, for each shard, each worker thread records the number of received requests during a fixed period (i.e., 500ms), and sends the statistic data to CM. Since `ROWAN` is one-sided and thus backups are unaware of replication writes, we let worker threads in primaries record the number of received replication writes for backup shards. CM calculates the load of each server according to these statistics. If a server has a load that is higher than the average load by a threshold (i.e., 30%), CM determines that the server is *overloaded*. CM produces a new shard distribution, where the hottest shards in overloaded servers are moved to underloaded servers, with a goal of making the load of every server within 5% of the average. Then, it saves a migration list in the config-



uration, which contains a triple  $\langle$ source server, target server, shard ID $\rangle$  for each migration task. Finally, CM increments the term, writes the new configuration (including the new shard distribution) to Zookeeper, and sends it to all servers.

Next, we describe how ROWAN-KV migrates a primary shard from a *source* server to a *target* server (migrating a backup shard is much easier since it does not serve client requests). Upon receiving the new configuration, servers cache it to local memory. From this point, the source server rejects client requests for the migrated shard. Clients will fetch the new configuration from CM when receiving rejected responses, so subsequent requests to the migrated shard will be sent to the target server. Then, the source server sends a message to the target server; the message contains the shard version of the migrated shard. Upon receiving both the message and the new configuration, the targeted server starts to serve requests for the migrated shard. In this way, ROWAN-KV guarantees that only one server can serve the shard at any given time. Then, the process of data migration starts:

- In the source server, a migration thread requests free PM segments from the target server via RPCs, traverses the index of the migrated shard, and stores the associated log entries to remote segments via RDMA WRITE.
- In the target server, a migration thread scans segments written by the source server and installs log entries in the shard's index. Upon a PUT request to the migrated shard, the target server handles it as normal. Upon a GET request, the target server searches the index; if the corresponding key is not found, the target server routes the GET request to the source server since some objects have not been migrated yet. Of note, the versions in log entries resolve the conflicts between the migration thread and concurrent PUT requests.

The target server informs CM when it finishes data migration. Then, CM deletes the migration task from the migration list and writes the new configuration to Zookeeper. Finally, CM sends a message to the source server to inform it to free the index of migrated shard; the associated log entries in the source server will be removed by garbage collection.

If the migration is interrupted due to failures of the source/target server, the CM first rolls back the shard distribution in the configuration to the state before migration. Then, the CM deletes the associated task in the migration list and performs the normal failover process. In addition, the CM informs the target server (if alive) to release resources allocated for the interrupted migration task (e.g., migration thread and index).

## 4.7 Cold Start

When the entire cluster experiences a power failure, ROWAN-KV can guarantee durability of data. Upon recovery, the CM fetches the configuration from Zookeeper, and disseminates it to all servers. Each server obtains the metadata for all its segments via the segment meta table (recall §4.2.1). For a shard, its primary extracts associated log entries from *using* segments whose *owner* is worker threads; then, the primary

sends these log entries to backups, to make all replicas own the same set of log entries. Each primary builds indexes for shards it manages by processing segments, and constructs valid shard versions. If two log entries have the same targeted key, the one with the larger version is more recent. Finally, ROWAN-KV resumes unfinished migration tasks according to the migration list stored in the configuration.

## 5 Implementation

We implement ROWAN-KV in Linux hosts. ROWAN-KV is a fully user-space system: it uses *libverbs* for RDMA operations and CPU memory instructions for accessing PM.

### 5.1 Threading Model

ROWAN-KV binds each thread (i.e., worker threads, clean threads, digest threads, and control thread) to an exclusive CPU core. ROWAN-KV follows two principles:

**Minimizing inter-thread communication.** First, each worker thread handles both network I/O and KV logic; this avoids request dispatch in systems that have dedicated threads to poll network requests [56], thus enjoying high multicore scalability. Second, a thread hands over segments to other threads *in a batch manner* (§4.3) using thread-safe queues.

**Avoiding thread blocking.** To avoid blocking due to waiting for network events, worker threads adopt a coroutine-like approach to interleave work: after issuing ROWAN operations for a PUT, a worker thread saves the context of the PUT request (e.g., the targeted key); then, it polls the RDMA completion queue, getting new requests to execute. Upon receiving ACKs from backups, the worker thread restores the PUT's context and continues the remaining logic. In this way, a worker thread can concurrently handle multiple PUT requests.

### 5.2 Network Components

**RPC.** ROWAN-KV uses an RPC framework for client-server and inter-server communication (not include replication). We build the RPC framework with RDMA SEND and RECV verbs using unreliable datagram (UD) QPs. Specifically, each worker thread creates a UD QP to receive requests and send responses. When a client joins the ROWAN-KV cluster, it establishes RPC connections with a worker thread in every server. Like FaSST [44], our RPC framework currently does not support messages larger than an MTU. To reduce CPU consumption on PM reads: the RPC framework leverages RNICs' scatter-gather DMA to gather RPC headers and PM-resident objects, generating responses of GET requests.

**ROWAN.** To realize ROWAN, every worker thread builds a reliable connection (RC) QP with every remote control thread.

At the sender side, a worker thread uses the associated send queue in QPs to issue ROWAN operations. A ROWAN operation contains a SEND followed by a 1B READ for persistence (§3). SEND and READ are sent in one `ibv_post_send` call. For a worker thread, all its ROWAN QPs and RPC QP share the same CQ, so that it can be aware of ROWAN ACKs and new RPC messages by polling the CQ. We mark SEND as un signaled to eliminate a completion event. For READ, we store the context



id of the associated PUT request (§5.1) into the `wr_id` field, so that worker threads can distinguish RowAN ACKs belonging to different PUT requests when polling the CQ.

At the receiver side, a control thread manages all RowAN QPs connected to remote worker threads; these QPs share an MP SRQ. The control thread pushes PM segments to the MP SRQ via RECV. We register PM to RNICs using physical addresses [64], to remove virtual-to-physical translation tables in RNICs and thus reduce cache thrash of RNICs.

**Mitigating the impact of disabled DDIO.** We disable DDIO to ensure the RNICs can land data to PM (rather than CPU cache). However, disabling DDIO will ① cause CPU cache miss when handling RPCs and ② degrade performance of DMA operations between RNICs and memory. For ①, worker threads poll multiple RPC messages at a time, and issue prefetch instructions to them. For ②, for RDMA READ used for persistence, we set its source address to RNICs' device memory [4, 68], to eliminate a DMA write at senders. We expect that DDIO does not need to be disabled, with next-generation RNICs supporting RDMA flush extensions [12].

### 5.3 Storage Components

**PM management.** We configure Optane DIMMs in App-Direct mode, which exposes PM as a range of physical memory. RowAN-KV splits the PM space into 4MB segments and stores the segment meta table in a predefined PM area (recall §4.2.1). A DRAM-resident free list records free segments, to serve segment allocation. We add padding for each log entry, making it 64B-aligned; it can ① avoid expensive PM read-modify-writes on receiver-side RNICs [70] when performing RowAN operations, and ② avoid slow repeated writes to the same cache lines [25, 42] in logs.

**DRAM indexes.** Each per-shard index is implemented with a concurrent bucket hash table [51]. The hash table is organized into a bucket array, where each bucket contains multiple 64-bit items. An item is composed of a 16-bit tag and a 48-bit PM address: the tag is a part of a key's hash value, to filter out mismatched searches and thus reduce PM reads; the PM address points to log entries. For a key, its targeted bucket is calculated by  $hash(key) \% sizeof(bucket\ array)$ . If the targeted bucket is full when inserting a key, threads create a new free bucket and link it to the targeted bucket, forming a bucket chain. Indexes support conditional update to resolve conflicts between threads: indexes omit an update if its log entry has version that is smaller than the one indexes are pointing to.

## 6 Evaluation

### 6.1 Experimental Setup

**Environment.** We use 6 machines as servers and 8 machines as clients. Each machine is equipped with the Intel Xeon Gold 6240M CPU (18 physical/36 logical cores), 96GB DRAM, and one 100Gbps Mellanox ConnectX-5 RNIC. All machines are connected to a 100Gbps Mellanox IB switch. Each server machine owns three 256GB Optane DIMMs (ADR mode).

Unless otherwise specified, we run RowAN-KV on 6 servers.

In each server, we use 24 cores for worker threads, 5 cores for digest threads, 6 cores for clean threads, and 1 core for control thread. The control thread also manages leases, with a lease time of 10ms. The CM and Zookeeper instance (3-way replication) run on client machines. Each client machine runs multiple client threads to issue requests to servers. We set the replication factor to 3. Each server holds 48 shards.

**Workloads.** We evaluate RowAN-KV using YCSB [26] with different PUT:GET ratios: **Load A** — 100% PUT (write-only); **A** — 50% PUT and 50% GET (write-intensive); **B** — 5% PUT and 95% GET (read-intensive); **C** — 100% GET (read-only). Key distribution follows Zipfian with parameter 0.99 (default parameter in YCSB). We populate 200 million objects into KVSs before each experiment. We use three Facebook workloads [24] to generate object size: **ZippyDB** (for general data store) — 90.8B average object size; **UP2X** (for AI/ML services) — 57.25B average object size; **UDB** (for social graph) — 153.8B average object size.

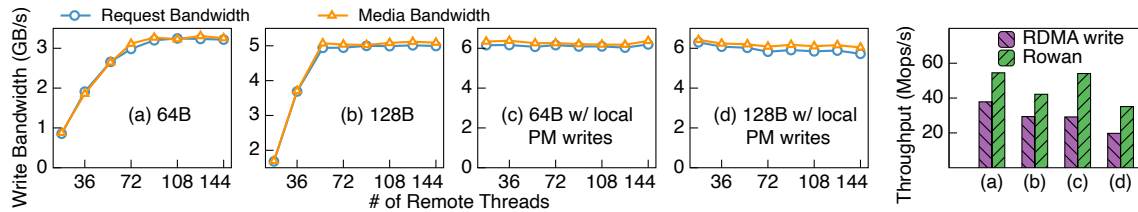
**Comparing targets.** We compare RowAN-KV with four KVSs, each using a specific replication approach:

- **RPC-KV.** It uses RPC to perform replication. Each server maintains per-thread b-logs, and primaries issue replication writes via RPC. Upon receiving a replication RPC, the worker thread appends the log entry into its local b-log.
- **RWrite-KV.** It uses FaRM's approach [31] to perform replication. Each worker thread has an exclusive remote b-log at every remote server. During replication, the worker thread issues WRITE for appending log entries to its b-logs at backups. Each server stores  $(m-1)*n$  b-logs, where  $m$  is the number of servers and  $n$  is the worker thread count.
- **Batch-KV.** Batch-KV is a variant of RWrite-KV and uses WRITE for replication. Each worker thread generates large-sized WRITE requests to its remote b-logs by *batching log entries*, to mitigate DLWA. To reduce latency, worker threads immediately send batched log entries to backups once 1) the total size is larger than an XPLine, i.e., 256B, or 2)  $5\mu s$  timeout is triggered.
- **Share-KV.** It is another variant of RWrite-KV and uses WRITE for replication. Worker threads in a server share the same remote b-log at a remote server, to reduce b-log count and thus mitigate DLWA. Worker threads use local atomic increment to obtain contiguous addresses in remote b-logs.

All systems are implemented in the same codebase (including optimizations in §5), to allow us to focus on the effects of replication approaches. By default, we disable DDIO to provide one-sided persistence. For RPC-KV, DDIO is enabled. We compare RowAN-KV with two off-the-shelf KVSs in §6.7.

### 6.2 RowAN Performance

We repeat the experiment in §2.4, to show performance of RowAN abstraction. Figure 8 presents the result of one RowAN instance. RowAN can largely eliminate DLWA in case of numerous concurrent remote small writes. The DLWA is less than 1.029 $\times$  when no local PM writes exist (Figure 8(a) and



**Figure 8:** ROWAN performance.  $DLWA = \text{media bandwidth}/\text{request bandwidth}$ . A number of threads issue 64B/128B writes to a remote server’s PM via a ROWAN instance. In (c) and (d), 18 CPU cores in the remote server perform local sequential PM writes.

(b)), and less than  $1.056\times$  when local PM writes exist (Figure 8(c) and (d)). This is because ROWAN can merge remote small writes into a single write stream, enabling efficient hardware combining in Optane DIMMs’ XPBuffer.

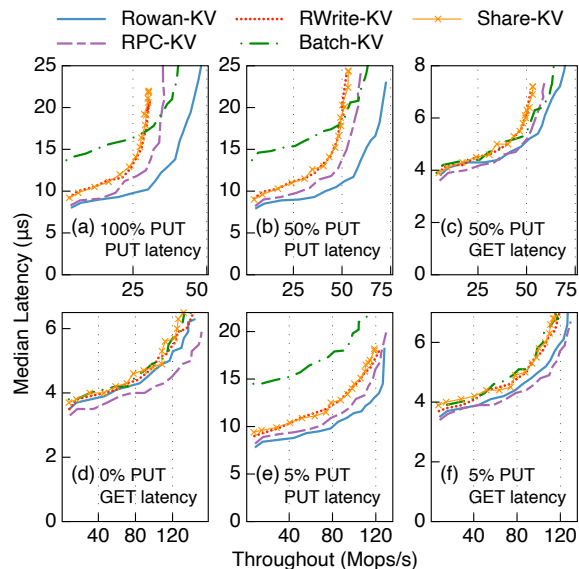
Further, we report the peak throughput of ROWAN and RDMA WRITE under these four cases, as shown in the rightmost subfigure of Figure 8. When no local PM writes exist, ROWAN can deliver 54.5 Mops/s for 64B remote PM writes and 42.2 Mops/s for 128B one, outperforming WRITE by  $1.44\times$  and  $1.43\times$ , respectively. When local PM writes appear, ROWAN outperforms WRITE by  $1.85\times/1.78\times$  for 64B/128B writes. Three causes make ROWAN performant. First, ROWAN largely eliminates DLWA, improving the available PM bandwidth. Second, on the data path of ROWAN, all PM writes are performed by the receiver-side RNIC, ensuring high throughput. Finally, on the control path, by leveraging ring CQ and MP SRQ, the control thread only performs very lightweight tasks, so it does not become the bottleneck. Of note, the bottleneck of ROWAN performance is 6GB/s PM write bandwidth in Figure 8(b)-(c), but processing capacity of RNICs in Figure 8(a). ROWAN does not achieve 75Mops/s (a maximal message rate that a 100Gbps RNIC can provide), since we disable DDIO and send an extra RDMA READ for each ROWAN operation.

### 6.3 ROWAN-KV Performance

Figure 9 shows median latency and throughput (6 servers) under YCSB workloads with ZippyDB object size. Since ROWAN-KV aims to accelerate replication, we report latency of PUT and GET separately. We increase the load generated by clients, and ensure that KVSs reach their peak throughput. We make two observations.

First, under read-only workloads (Figure 9(d)), RPC-KV has 5% higher throughput against other KVSs. This is because for KVSs using WRITE or ROWAN, DDIO is disabled, lowering RPC performance. Such performance gap can be eliminated with next-generation RNICs supporting RDMA flush extensions [12]. Under read-intensive workloads (Figure 9(e) and (f)), RPC-KV and ROWAN-KV have the similar throughput, since RPC-KV consumes CPU cycles of backups for 5% PUT requests, offsetting the benefits of DDIO. Compared with RPC-KV, ROWAN-KV has  $1.09\times$  lower median PUT latency due to elimination of backups’ software queueing, and  $1.27\times$  higher median GET latency due to disabled DDIO.

Second, under write-only and write-intensive (i.e., 50% PUT) workloads (Figure 9(a)-(c)), ROWAN-KV has the highest throughput with the lowest median latency. We compare



**Figure 9:** Median latency vs. throughput. ZippyDB object size. We report PUT latency and GET latency separately.

ROWAN-KV with the other four KVSs in turn.

With RPC-KV. ROWAN-KV achieves peak throughput of  $72.7/48.2\text{Mops/s}$  under write-intensive/write-only workloads, outperforming RPC-KV by  $1.22\times/1.37\times$ . This is because ROWAN-KV replicates log entries via *one-sided* ROWAN, saving CPU cycles that handle replication RPCs. The saved CPU cycles can be used for primaries to handle RPCs from clients. At the peak throughput of RPC-KV, ROWAN-KV has  $1.77\times/1.61\times$  lower median PUT latency under write-intensive/write-only workloads. This is because compared with RPCs, one-sided ROWAN eliminates backup-side software queueing/execution on the critical path of replication. Avoiding replication RPCs also makes ROWAN-KV reduce median GET latency by 23%. Figure 10 shows DLWA of write-only and write-intensive workloads (6 servers). For ROWAN-KV and RPC-KV, the DLWA is less than  $1.032\times$ . This is because they generate a small number of PM write streams: in each server, ROWAN-KV has 24 t-logs and 1 b-log; RPC-KV has 24 t-logs and 24 b-logs (recall we use 24 worker threads in experiments). Optane DIMMs can efficiently combine adjacent small writes of the same logs into XPLINE writes, when write stream count is not high (recall Figure 2(c) and (d)).

With RWrite-KV. Compared to RWrite-KV, ROWAN-KV yields  $1.39\times/1.61\times$  higher throughput and  $2.06\times/2.1\times$  lower median PUT latency under write-intensive/write-only work-

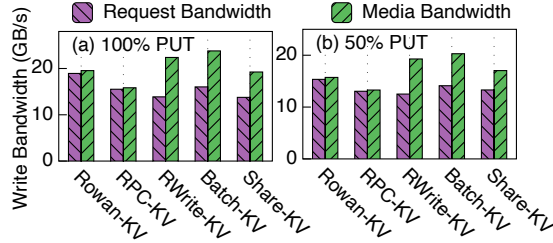


Figure 10: DLWA (6 servers) at peak throughput.

loads. The main culprit of RWrite-KV’s low performance is DLWA: as shown in Figure 10(a), it suffers 1.54× DLWA.

This is because RWrite-KV owns lots of logs (i.e., 24×6 in experiments) in a server to accommodate small writes, exceeding the combining capacity of Optane DIMMs: a large number of write streams are equivalent to random writes. In RWrite-KV, Optane DIMMs trigger lots of read-modify-write events, which squander a considerable number of hardware resources (e.g., XPBuffer), degrading performance of PM accesses. To demonstrate it, we measure the latency of remote persistence operations of ROWAN-KV and RWrite-KV under write-intensive workloads. Figure 11 shows the latency distribution. Remote persistence in RWrite-KV is slow (against ROWAN-KV), with 11.5μs median latency and 24μs 99% latency. Of note, although RNICs are ideally capable of providing an RTT of ~2μs, the 6.6μs median latency of ROWAN is reasonable, since 1) we disable DDIO and each ROWAN operation contains a synchronous RDMA READ, and 2) the latency is measured under high loads where RNICs suffer from DMA queuing.

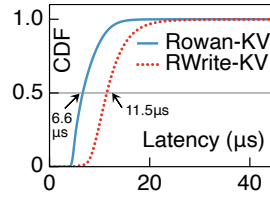


Figure 11: Latency CDF.

With Batch-KV. Batch-KV boosts the throughput of RWrite-KV by 1.23×/1.35× under write-intensive/write-only workloads, since it reduces the number of WRITE and mitigates DLWA (by 12%) via batching. However, batching makes Batch-KV suffer the highest PUT latency among all KVSs: even under low loads, Batch-KV has more than 50% higher PUT latency compared with ROWAN-KV. In terms of throughput, ROWAN-KV outperforms Batch-KV by 1.13×/1.19× under write-intensive/write-only workloads. This is because Batch-KV still experiences DLWA: it frequently fails to accumulate enough small writes within 5μs timeout for two reasons: 1) All GET requests do not generate writes but consume CPU time; 2) Only writes to the same destination can be batched; yet, due to sharding of KVSs, for a server acting as primaries, the backups of its shards are distributed to *multiple* servers, greatly decreasing the batching opportunity. We also change the timeout value to 20μs, and Batch-KV delivers 9% lower throughput against ROWAN-KV, with unacceptable latency.

With Share-KV. Share-KV reduces DLWA of RWrite-KV by 26%/22% under write-intensive/write-only workloads, since it lets worker threads share the same b-logs. However, it still

	ROWAN-KV	RPC-KV	RWrite-KV	Batch-KV	Share-KV
UP2X	73.9Mops/s	61.5Mops/s	56.2Mops/s	70.3Mops/s	56.0Mops/s
UDB	62.5Mops/s	50.4Mops/s	49.9Mops/s	57.1Mops/s	50.6Mops/s

Table 2: Throughput under write-intensive workloads.

suffers sizable DLWA (1.28×~1.39×), resulting in lower performance against ROWAN-KV. This is because although worker threads in a Share-KV server generate contiguous remote addresses for WRITE, the asynchronous network makes receiver-side RNICs receive and perform these writes in an out-of-order manner. In contrast, for ROWAN-KV, leveraging ROWAN, receiver-side RNICs decide destination addresses of writes. Besides, ROWAN can merge writes from *different* servers.

**Tail latency.** Under write-intensive workloads with 50Mops/s throughput, ROWAN-KV’s 99% latency is 20.5μs, which is 1.26×, 2.11×, 1.53×, and 1.87× lower than that of RPC-KV, RWrite-KV, Batch-KV, and Share-KV, respectively.

**Performance under uniform workloads.** We evaluate ROWAN-KV using uniform key distribution. ROWAN-KV delivers 67.86Mops/s and 108.19Mops/s in cases of 50% PUT and 5% PUT, respectively, which are 6.6% and 15.5% slower than throughput of Zipfian skewed workloads (see Figure 9). ROWAN-KV has higher performance under skewed workloads for two reasons. First, in our cluster of 6 servers, due to hash-based sharding, there is no observable load imbalance across servers under skewed workloads. Second, threads enjoy much better cache locality under skewed workloads.

**Performance with UP2X/UDB object size.** Due to space limitations, here we only report the throughput under write-intensive workloads, as shown in Table 2. ROWAN-KV delivers the highest throughput via powerful ROWAN abstraction.

## 6.4 Sensitivity Analysis

We conduct experiments on sensitivity analysis using write-intensive workloads and ZippyDB object size.

**Impact of object size.** We change object size to generate varying log entry size. As shown in Figure 13(a), when log entry size is an integer multiple of XPLine size (e.g., 256B), all KVSs do not induce severe DLWA; thus, ROWAN-KV and KVSs using WRITE have the similar throughput. RPC-KV consumes CPU cycles for replication RPCs, so it has 21% lower throughput against ROWAN-KV with 1024B log entries.

**Impact of replication factor.** Figure 13(b) presents throughput with varying replication factor. As replication factor increases, performance improvement between ROWAN-KV and other KVSs increases. This is because, with higher replication factors, RPC-KV needs to consume more CPU cycles to handle a PUT request, and WRITE-enabled KVSs issue more WRITE and thus induce more DLWA. In contrast, ROWAN-KV replicates objects in a one-sided manner and merges all remote writes into a single b-log in a sequential manner.

**Impact of worker thread count.** Figure 13(c) presents throughput with different worker thread counts. We make two observations. First, when the number of threads is small (i.e., ≤ 16), RPC-KV has the lowest throughput, since the CPU



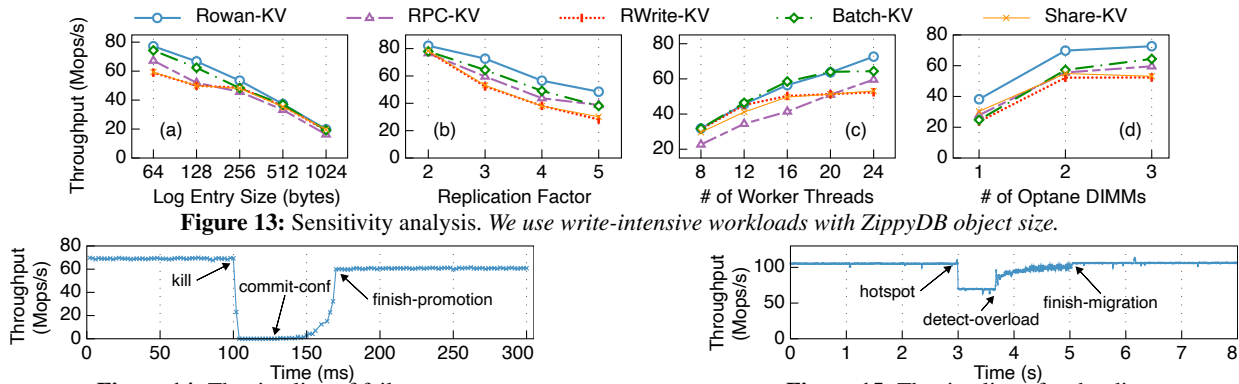


Figure 13: Sensitivity analysis. We use write-intensive workloads with ZippyDB object size.

Figure 14: The timeline of failover.

Figure 15: The timeline of resharding.

is the bottleneck. Second, RWrite-KV and its variants yield poor scalability. This is because 1) for RWrite-KV and Batch-KV, the number of b-logs is proportional to thread count, and 2) for Share-KV, RNICs are more likely to receive and perform WRITE to b-logs in an out-of-order manner in case of high thread count; thus, they suffer more severe DLWA with higher thread count. DLWA harms PM performance (recall Figure 11), thus stalling throughput. In contrast, ROWAN-KV exhibits superior throughput with different thread counts.

**Impact of PM bandwidth.** Figure 13(d) presents throughput with different number of Optane DIMMs per server. In case of one Optane DIMM, the PM bandwidth is bottleneck. Thus, RWrite-KV (which has the most severe DLWA) is outperformed by ROWAN-KV, RPC-KV, Batch-KV, and Share-KV by 1.61 $\times$ , 1.18 $\times$ , 1.05 $\times$ , and 1.28 $\times$ , respectively. In case of three Optane DIMMs, CPU becomes the bottleneck and limits throughput, and PM bandwidth is not saturated (see Figure 10). ROWAN-KV squeezes out CPU resources in two aspects: 1) it reduces CPU involvement via ROWAN’s one-sided semantic; 2) it largely eliminates DLWA, streamlining Optane DIMMs’ internal operations and thus improving persistence efficiency of worker threads.

### 6.5 Failover and Cold Start

**Failover.** We kill a server to test ROWAN-KV’s failover mechanism. We use write-intensive workloads with ZippyDB objects and ROWAN-KV runs for 50 seconds before the test. Figure 14 shows the timeline, where throughput is recorded per 2ms. The server is killed at time 100ms (i.e., `kill`). ROWAN-KV uses 26ms to commit the new configuration (i.e., `commit-conf`), which mainly includes detecting failure (8ms), writing new configuration to Zookeeper (4.3ms), and waiting for the failed server’s lease to expire (10ms). Then, ROWAN-KV consumes about 44ms to promote backups to primaries (i.e., `finish-promotion`). At this point, ROWAN-KV can serve all requests from clients.

**Cold start.** We test cold start of a ROWAN-KV instance, which contains 10 billion ZippyDB objects and thus occupies about 3TB PM space (6 servers). The time of cold start is 49.6s. Although cold start is slow, it is not common in datacenters. Periodically checkpointing DRAM-resident indexes can accelerate cold start, and we leave it for future work.

### 6.6 Dynamic Resharding

In this experiment, we evaluate ROWAN-KV’s dynamic resharding mechanism. We use read-intensive workloads with ZippyDB objects. Figure 15 presents the total throughput (6 servers) over time. At first, clients generate a uniform workload and each server has a similar CPU utilization (i.e., 90.2% ~ 90.9%). At time 3s (i.e., `hotspot`), clients shift 80% requests for server A to a shard residing on server B, to make server B have a hotspot shard and overloaded. The throughput drops by 33% due to load imbalance. At this time, server A and server B have a CPU utilization of 60.7% and 91% respectively. The average CPU utilization of the other 4 servers drops to 72.8%, since requests to overloaded server B suffer from long queueing and thus the limited number of clients cannot generate enough requests to other servers. CM detects the overload after 660ms (i.e., `detect-overload`) and produces a migration task that migrates the hotspot shard from server B to server A. The migration takes 1346ms and moves about 1.1 million objects. The throughput increases as the migration proceeds, since more GET requests to the hotspot shard can be served by server A. Finally, the system achieves a load-balanced state with steady throughput.

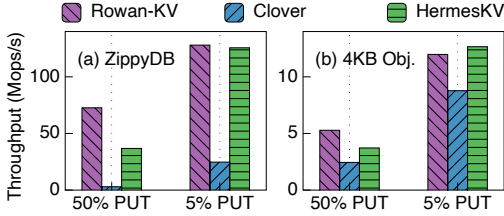
### 6.7 Comparison with Other Systems

We compare ROWAN-KV with two state-of-the-art replicated KVSs designed for RDMA networks:

- **Clover** [63]. Clover runs on disaggregated PM, where PM servers do not have compute resources. Clients perform GET operations via RDMA READ verbs, and perform PUT operations (including replication) using a combination of RDMA WRITE and ATOMIC.
- **HermesKV** [45]. It is a DRAM-resident KVS built on Hermes [45], a broadcast-based replication protocol. HermesKV uses RPC for all inter-server communication (including replication). We modify the code to support PM: we store objects in PM and issue `ntstore` instructions for durability; indexes are in DRAM. In addition, we let clients generate KV requests to HermesKV servers.

We use ZippyDB objects and 4KB objects to test KVSs under small writes and large writes, respectively. The key distribution follows Zipfian with parameter 0.99. The replication factor is 3 and HermesKV runs with enabled DDIO.





**Figure 16:** Comparison with Clover and HermesKV. (a) Throughput with ZippyDB objects. (b) Throughput with 4KB objects.

Figure 16(a) shows the results of small writes (ZippyDB objects). Under write-intensive workloads (i.e., 50% PUT), ROWAN-KV outperforms Clover and HermesKV by 24.5 $\times$  and 1.98 $\times$ , respectively. Two reasons contribute to Clover’s low throughput. First, due to the disaggregated architecture, every operation in Clover needs multiple network communications. Second, Clover uses RDMA ATOMIC to resolve conflicts between client threads, which leads to significant performance degradation when contention appears [35]. Using RDMA ATOMIC on PM is also considered slow due to its read-modify-write behavior [70]. HermesKV uses RPC for replication which consumes CPU cycles at backups, so it is outperformed by ROWAN-KV which uses one-sided ROWAN for replication. We measure DLWA of these KVSSs. Clover has 1.86 $\times$  DLWA and HermesKV has 2.95 $\times$  DLWA, since both of them generate a large number of random small writes on PM: for a PUT operation, Clover performs copy-on-write using WRITE and HermesKV performs in-place updates. In contrast, ROWAN-KV adopts the log-structured approach to manage objects and exploits ROWAN abstraction to minimize the number of write streams; thus, DLWA of ROWAN-KV is less than 1.032 $\times$ . Under read-intensive workloads (i.e., 5% PUT), ROWAN-KV and HermesKV have similar throughput, which far exceeds that of Clover (about 5 $\times$ ).

Figure 16(b) reports the results of large writes (4KB objects). Under write-intensive workloads, ROWAN-KV outperforms HermesKV by 1.42 $\times$  and is bottlenecked by PM write bandwidth. HermesKV can not approach the limitation of PM write bandwidth, since its backups waste lots of CPU cycles to copy/persist large objects from RPC buffers to PM. Under read-intensive workloads, ROWAN-KV and HermesKV are bottlenecked by the network bandwidth (11GB/s per server), which is much lower than PM read bandwidth (18GB/s).

## 7 Discussion

Although Intel killed Optane memory business for commercial reasons in summer 2022, we believe that ROWAN is still applicable to future byte-addressable storage devices. For example, CXL storage devices (e.g., Samsung’s Memory-Semantic SSD [13]), which are considered promising alternatives to Optane DIMMs, share similarities with Optane DIMMs: 1) limited write bandwidth; 2) byte interfaces with a block-level internal access granularity (e.g., flash page). Thus, when many remote threads concurrently access CXL storage devices with small IO size, ROWAN can still effectively mitigate DLWA and thus boost system performance.

## 8 Related Work

**PM KVSSs.** There are a host of works on PM KVSSs, but most of them are single-machine (except Clover [63]). HiKV [71] and Bullet [39] are designed before the availability of real PM devices; both of them store objects into fine-grained PM hash tables. However, real PM devices have block-level internal access granularity (e.g., 256B in Optane DIMMs). To reduce DLWA, recent PM KVSSs, including FlatStore [25], Viper [22], and Pacman [66], adopt log-structured approaches to manage objects. ROWAN-KV also uses log-structured approach for the same reason, but focuses on distributed environments where objects are sharded and replicated.

**RDMA replication.** RDMA replication can be categorized into two groups, namely *backup-active* and *backup-passive*, depending on whether backups consume CPUs on the critical path of replication. Lots of systems [20, 21, 41, 45, 65] belong to backup-active group, where backups’ CPUs need to process messages during replication. For backup-passive group [17, 31, 46, 47, 57, 62, 69, 75], primaries only need to wait for ACKs from the RNIC hardware of backups. For example, Hyperloop [47] uses RDMA WAIT and WRITE verbs to realize chain replication. ROWAN-KV belongs to the backup-passive group, so it features low latency and high CPU efficiency. Yet, traditional backup-passive approaches can lead to DLWA on PM KVSSs, driving us to design the ROWAN abstraction.

**RDMA abstraction.** Due to limited expressivity of RDMA verbs, several works propose new RDMA abstractions [18, 19, 23, 27, 60, 72]. StRoM [60] and RMC [19] allow applications to define functions on NICs. Aguilera et al. [18] and PRISM [23] propose several new RDMA verbs to support far memory data structures and distributed systems. RedN [58] makes RDMA Turing complete using self-modifying chains. All above works (except RedN) require RNIC modification or specialized hardware (e.g., SmartNICs). In contrast, ROWAN can be realized with off-the-shelf RNICs, leveraging RNIC features such as SRQ and MP RQ. Besides, ROWAN targets handling high fan-in small PM writes.

## 9 Conclusion

This paper explored how to efficiently replicate PM KVSSs using RDMA. We showed that existing approaches using RDMA WRITE cause severe device-level write amplification (DLWA) on PM. To this end, we proposed ROWAN, a one-sided RDMA abstraction that can merge numerous remote writes into a single stream. Based on ROWAN, we built ROWAN-KV, a log-structured PM KVSS; it outperforms RPC and RDMA WRITE alternatives in throughput and latency under write-intensive workloads, while achieving low DLWA.

## Acknowledgements

We sincerely thank our shepherd Bernard Wong for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Natural Science Foundation of China (Grant No. 61832011, 62022051) and Alibaba Group through AIR Program.

## References

- [1] Apache Cassandra Data Partitioning. <https://www.instaclustr.com/blog/cassandra-data-partitioning/>, 2022.
- [2] Choosing the Right DynamoDB Partition Key. <https://aws.amazon.com/en/blogs/database/choosing-the-right-dynamodb-partition-key/>, 2022.
- [3] Data Distribution and Movement in FoundationDB. <https://github.com/apple/foundationdb/wiki/Data-Distribution-and-Movement>, 2022.
- [4] Device Memory of RNICs. [https://man7.org/linux/man-pages/man3/ibv\\_alloc\\_dm.3.html](https://man7.org/linux/man-pages/man3/ibv_alloc_dm.3.html), 2022.
- [5] Intel Data Direct I/O Technology. <https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html>, 2022.
- [6] Intel Optane DC Persistent Memory Module (PMM). <https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm/>, 2022.
- [7] Introduce Verbs API for Multi-packet Work Request. <https://marc.info/?l=linux-rdma&m=151311334131294&w=2>, 2022.
- [8] ipmctl. <https://github.com/intel/ipmctl>, 2022.
- [9] Multi-Packet RQ. <https://docs.mellanox.com/display/rdmacore50/Multi-Packet+RQ>, 2022.
- [10] Partitioning and horizontal scaling in Azure Cosmos DB. <https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview>, 2022.
- [11] RDMA Aware Networks Programming User Manual. [https://www.mellanox.com/related-docs/prod\\_software/RDMA\\_Aware\\_Programming\\_user\\_manual.pdf](https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf), 2022.
- [12] RDMA Verbs Extensions for Persistency and Consistency. [https://www.snia.org/sites/default/files/SDC/2016/presentations/persistent\\_memory/IdanBurststein\\_RDMA\\_VERBs\\_Extensions.pdf](https://www.snia.org/sites/default/files/SDC/2016/presentations/persistent_memory/IdanBurststein_RDMA_VERBs_Extensions.pdf), 2022.
- [13] Samsung Electronics Unveils Far-Reaching, Next-Generation Memory Solutions at Flash Memory Summit 2022. <https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022>, 2022.
- [14] Twitter’s doubling of character count from 140 to 280 had little impact on length of tweets. <https://techcrunch.com/2018/10/30/twitters-doubling-of-character-count-from-140-to-280-had-little-impact-on-length-of-tweets/>, 2022.
- [15] ZippyDB: the Architecture of Facebook’s Strongly Consistent Key-Value Store. <https://www.infoq.com/news/2021/09/facebook-zippydb/>, 2022.
- [16] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.
- [17] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.
- [18] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS ’19*, page 120–126, New York, NY, USA, 2019. Association for Computing Machinery.
- [19] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote Memory Calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets ’20*, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery.
- [20] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.
- [21] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. RDMC: A Reliable RDMA Multicast for Large Objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82, 2018.

- [22] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.*, 14(9):1544–1556, May 2021.
- [23] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 228–242, New York, NY, USA, 2021. Association for Computing Machinery.
- [24] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.
- [25] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.
- [26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [27] Alexandres Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. SABRes: Atomic Object Reads for in-Memory Rack-Scale Computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49*. IEEE Press, 2016.
- [28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.
- [29] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.
- [30] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.
- [31] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.
- [32] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020.
- [33] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.
- [34] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiayi Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021.
- [35] Stewart Grant and Alex C Snoeren. In-network Contention Resolution for Disaggregated Memory.
- [36] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.*, 14(4):626–639, December 2020.
- [37] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

- [38] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuai Peng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.
- [39] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, July 2018. USENIX Association.
- [40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC’10*, page 11, USA, 2010. USENIX Association.
- [41] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.*, 36(2), apr 2019.
- [42] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC ’20*, page 105–119, New York, NY, USA, 2020. Association for Computing Machinery.
- [43] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.
- [44] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’16*, page 185–201, USA, 2016. USENIX Association.
- [45] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.
- [46] Mikhail Kazhemiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, and Khuzaima Daudjee. Sift: Resource-Efficient Consensus with RDMA. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, CoNEXT ’19*, page 260–271, New York, NY, USA, 2019. Association for Computing Machinery.
- [47] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.
- [48] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.
- [49] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.
- [50] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 553–569, New York, NY, USA, 2021. Association for Computing Machinery.
- [51] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.
- [52] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S.



- Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.
- [53] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.
- [54] Jinyoung Oh and Youngjin Kwon. *Persistent Memory Aware Performance Isolation with Dicio*, page 97–105. Association for Computing Machinery, New York, NY, USA, 2021.
- [55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, NSDI'19*, page 361–377, USA, 2019. USENIX Association.
- [56] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAM-Cloud Storage System. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.
- [57] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '15*, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.
- [58] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *Proceedings of NSDI'22*, Apr 2022.
- [59] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. TH-DPMS: Design and Implementation of an RDMA-Enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage*, 16(4), oct 2020.
- [60] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarini, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20*, New York, NY, USA, 2020. Association for Computing Machinery.
- [61] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20*, New York, NY, USA, 2020. Association for Computing Machinery.
- [62] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 851–863, Boston, MA, July 2018. USENIX Association.
- [63] Shin-Yeh Tsai, Yizhou Shan, and Yiyang Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.
- [64] Shin-Yeh Tsai and Yiyang Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.
- [65] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17*, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.
- [66] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 773–788, Carlsbad, CA, July 2022. USENIX Association.
- [67] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.
- [68] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22*, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.
- [69] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better. In *Proceedings of the*

*13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18*, page 233–251, USA, 2018. USENIX Association.

- [70] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536. USENIX Association, July 2021.
- [71] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 349–362, USA, 2017. USENIX Association.
- [72] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 111–125, Santa Clara, CA, February 2020. USENIX Association.
- [73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.
- [74] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21*, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.
- [75] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, page 3–18, New York, NY, USA, 2015. Association for Computing Machinery.
- [76] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD/PODS '21*, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.



# eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs

Jaehong Min<sup>1</sup>, Chenxingyu Zhao<sup>1</sup>, Ming Liu<sup>2</sup>, and Arvind Krishnamurthy<sup>1</sup>

<sup>1</sup>University of Washington

<sup>2</sup>University of Wisconsin-Madison

## Abstract

Emerging Zoned Namespace (ZNS) SSDs, providing the coarse-grained zone abstraction, hold the potential to significantly enhance the cost-efficiency of future storage infrastructure and mitigate performance unpredictability. However, existing ZNS SSDs have a static zoned interface, making them in-adaptable to workload runtime behavior, unscalable to underlying hardware capabilities, and interfering with co-located zones. Applications either under-provision the zone resources yielding unsatisfied throughput, create over-provisioned zones and incur costs, or experience unexpected I/O latencies.

We propose eZNS, an elastic-zoned namespace interface that exposes an adaptive zone with predictable characteristics. eZNS comprises two major components: a zone arbiter that manages zone allocation and active resources on the control plane, a hierarchical I/O scheduler with read congestion control, and write admission control on the data plane. Together, eZNS enables the transparent use of a ZNS SSD and closes the gap between application requirements and zone interface properties. Our evaluations over RocksDB demonstrate that eZNS outperforms a static zoned interface by 17.7% and 80.3% in throughput and tail latency, respectively, at most.

## 1 Introduction

The NVMe Zoned Namespace (ZNS) is a newly-introduced storage interface and has received significant attention from data center and enterprise storage vendors. By dividing the SSD physical address space into logical zones, migrating from device-side implicit garbage collection (GC) to host-side explicit reclaim, and eradicating random write accesses, a ZNS SSD significantly reduces device DRAM needs, resolves the write amplification (WAF) issue, minimizes costly overprovisioning, and mitigates I/O interference. However, the performance characteristics of the ZNS interface are not well-understood. In particular, to build efficient I/O stacks over it, we should be cognizant of (1) how the underlying SSD exposes the zone interface and enforces its execution restrictions; (2) what trade-offs the device's internal mechanisms make to balance between cost and performance. For

example, the device-enforced zone placement makes the actual I/O bandwidth capacity of a zone contingent on how a ZNS SSD allocates zone blocks across channels/dies. Further, a zone is not a performance-isolated domain, and one could observe considerable I/O interference for inter-zone read and write requests. Therefore, there is a strong need to understand its idiosyncratic features and bring enough clarity to storage applications.

We perform a detailed performance characterization of a commodity ZNS SSD, investigate its device-internal mechanisms, and analyze the benefits and pitfalls under different I/O profiles in both standalone and co-located scenarios. Using carefully calibrated microbenchmarks, we examine the interaction between zones and the underlying SSD from three perspectives: zone striping, zone allocation, and zone interference. We also compare with conventional SSDs when necessary to investigate the peculiarity of a ZNS SSD. Our experiments highlight the interface's capabilities to mitigate the burden on I/O spatial and temporal management, identify constraints that would cause sub-optimal performance, and provide guidance on overcoming the limitations.

We then propose eZNS, a new interface layer, which provides a device-agnostic zoned namespace to the host system, mitigates inter-/intra-zone interference, and improves the device bandwidth by allocating active resources based on the application workload profile. eZNS is transparent to upper-layer applications and storage stacks. Specifically, eZNS comprises two components: the zone arbiter on the control plane and a tenant-cognizant I/O scheduler on the data plane. The zone arbiter maintains the device shadow view that manages zone allocations and realizes a dynamic resource allocation by a zone ballooning mechanism. It allows serving applications to max out the device capability by enabling the maximum device parallelism given the workload profile and rebalancing inactive bandwidth across namespaces. The I/O scheduler of eZNS leverages the intrinsic characteristics of ZNS, where there are no hardware-hidden internal bookkeeping operations. Read I/Os become more predictable, and one can directly harness this property to examine inter-zone interference.



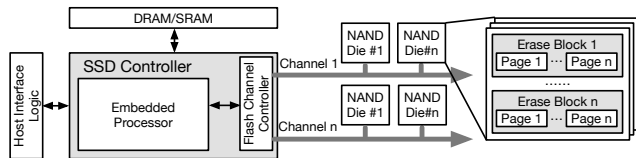


Figure 1: The architecture of a conventional and ZNS SSD.

On the other hand, write I/Os share a performance domain due to the write cache architecture of the SSD, causing global congestion across all active zones. eZNS, therefore, applies a local congestion control for reads and a global admission control for writes. Our I/O schedulers mitigate the interference independently but improve overall system performance cooperatively. We demonstrate benefits in the evaluation (§5) over micro-benchmarks and RocksDB.

## 2 Background and Motivation

This section reviews the basics of NAND-based SSDs, introduces the ZNS SSD and its features, and discusses the problems with the existing zoned interface.

### 2.1 NAND-based SSDs

A NAND-based SSD combines an array of flash memory dies and is able to deliver a bandwidth of several GB/s. It comprises four main architectural components (Figure 1): (1) a host interface logic (HIL) that implements the protocol used to communicate with the host, such as SCSI [40] and recent NVMe [29]; (2) an SSD controller, enclosing an embedded processor and a flash channel controller, which is responsible for the address translation and scheduling, as well as flash memory management; (3) onboard DRAM, buffering transmitted I/O data and metadata, storing the address translation table, and providing a write cache; (4) a multi-channel subsystem that connects NAND dies via a high-bandwidth interconnect. As shown in Figure 1, a NAND *die* consists of hundreds of *erase blocks*, where each *block* contains hundreds to thousands of *pages*. Each page encloses a fixed-sized data region and a metadata area that stores ECC and other information. Flash memory supports three major operations: *read*, *program*, and *erase*. The access granularity of a read/program is a page, while the erase command is performed in units of blocks. NAND flash memory has three unique characteristics [1, 10, 12, 19, 26]: (1) no in-place update, where the whole block must be erased before updating any page in that block; (2) asymmetric performance between reads and programs; (3) limited lifetime (endurance) – each cell has a finite number of program/erase (P/E) cycles [22].

To effectively use the NAND flash and address its limitations, SSDs employ a special mapping layer called the flash translation layer (FTL). It provides three major functionalities [13, 20, 33, 52]: (1) dynamically mapping logical block addresses (LBA) to physical NAND pages addresses (PPA); (2) implementing a garbage collection (GC) mechanism to

handle the no in-place update issue and asynchronously reclaim invalid pages; (3) applying a wear-leveling technique to evenly balance the usage (or aging property) of all blocks and prolong the SSD lifespan. However, FTL brings in considerable overheads. First, the translation table requires a large amount of DRAM to store the mapping entries, e.g., 1GB for 1TB NAND capacity for 4KB data unit size. Second, when serving a user I/O, the compounding effect of GC and wear-leveling would trigger additional SSD internal writes (i.e., copying valid pages to erase the block) and lead to the WAF (Write Amplification Factor) problem. Third, the FTL does not employ performance isolation mechanisms and incurs significant interference issues under mixed I/O profiles [28, 32].

### 2.2 Zoned Namespace SSDs

ZNS SSDs, a successor to Open-Channel (OC) SSDs [6, 9], have recently been developed to overcome the aforementioned limitations of conventional SSDs. There are several commodity ZNS SSDs from various vendors [34, 37, 38, 50]. A ZNS SSD applies the same architecture as a conventional one (Figure 1) but exposes the zoned namespace interface. A namespace is a separate logical block address space, like a traditional disk partition, but managed by the NVMe device controller rather than the host software. The device may control the internal block allocation of namespaces to optimize the performance based on the device-specific architecture. In ZNS SSD, the namespace comprises multiple zones instead of blocks in the conventional one, and each namespace owns dedicated *active resources* that are used to open and write a zone.

A ZNS SSD divides the logical address space of namespaces into fixed-sized zones, where each one is a collection of erase blocks and must be written sequentially and reset explicitly. ZNS SSDs present three benefits: (1) Maintain coarse-grained mappings between zones and flash blocks and apply wear-leveling at the zone granularity, requiring much smaller internal DRAM; (2) Eliminate the device-side GC and reclaim NAND blocks via explicit zone resets by host applications, which mitigates the WAF and log-on-log [51] issues and minimizes the over-provisioning overhead; (3) Enable the placement of opened zones across different device channels and dies, providing isolated I/O bandwidth and eliminating inter-zone write interference.

A zone has six states (i.e., *empty*, *implicitly open*, *explicitly open*, *closed*, *full*, *read only*, and *offline*). State transitions are triggered by either write I/Os or zone management commands (i.e., RESET, OPEN, CLOSE, and FINISH). A zone must be opened before issuing writes, but it is capable of serving reads in any state except the *offline* state. *closed* and *open* (both implicit and explicit) are *active* states that require the device to maintain NAND metadata for incoming user write I/Os, limiting the maximum number of active zones. SSDs employ the write cache in DRAM to align the wide range of user I/O sizes to the NAND program unit and comply with the NAND-

specific requirements (timings and program order). In case of a sudden power-off failure, the device flushes uncommitted data in the cache using batteries or capacitors as an emergency power source [46, 54]. Since active zones must have a buffer backed by energy devices for at least one NAND program unit in the cache, the maximum number of active zones is also constrained by the size of the write cache.

A zone provides three I/O commands: *read*, *sequential write*, and *append*. The *append* works similarly to the nameless write [53] but improves the host I/O efficiency rather than the internal NAND page allocation. Compared with the normal write, a zone *append* command does not specify the LBA in the I/O submission request, whilst the SSD will determine it at processing time and return the address in the response. Thus, user applications can submit multiple outstanding operations simultaneously without violating the restriction of sequential writes. Random writes are disallowed on ZNS SSDs, and the zone is erased as a whole (via the RESET). A ZNS SSD delegates the FTL and GC responsibilities to user applications, where they are performed at the zone granularity, thus eliminating traditional SSD overheads.

### 2.3 Small-zone and Large-zone ZNS SSDs

Zones can be classified into two types: *physical zone* and *logical zone*. Physical zones are the smallest unit of zone allocation and consist of one or more erasure blocks on a single die. They are device-backed and offer fine-grained control over storage resources. In contrast, logical zones refer to a striped zone region consisting of multiple physical zones. They can be implemented by either the device firmware or application and provide higher bandwidth through striping. Large-zone ZNS SSDs provide coarse-grained large logical zones with a fixed striping configuration that spans multiple dies across all internal channels but offers limited flexibility for controlling device behavior from the host software. This simplifies zone allocation but exposes a small number of active zones available for allocation to applications (e.g., 14 zones [50]). As a result, large-zone SSDs are more suitable for scenarios with small numbers of tenants, where the number of active zones required is not high. In addition, the application-agnostic fixed striping configuration does not adapt to workload profiles, resulting in low bandwidth utilization. Small-zone ZNS SSDs operate under similar hardware constraints but expose finer-grained physical zones. Each zone is contained within a single die but sufficiently large to encompass at least one erasure block. Small-zone SSDs provide greater flexibility and much more active resources (e.g., 256 zones in our testbed ZNS SSD) to support more I/O streams. In addition to increased flexibility, small-zone SSDs reduce the need for application-level garbage collection, especially while managing large numbers of small objects. Recent studies also corroborate some of these points. Specifically, Bae et al. [3] advocate a zone to be as small as possible to reduce the interference caused by high zone-reclaiming latencies. ZNS+ [16]

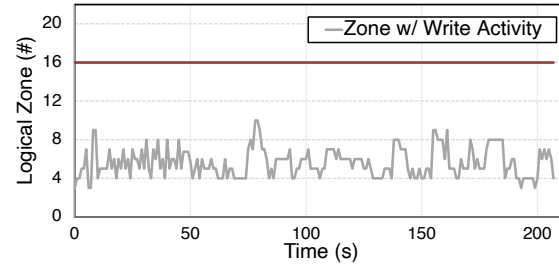


Figure 2: The number of zone with actual write activity when running the *fill-random* workload over the RocksDB. The storage backend is ZenFS. The maximum number of active zones is 16 (red line).

also prefers small zones as it minimizes the latency of COPY operations performed frequently in its F2FS implementation.

### 2.4 The Problem: Lack of an Elastic Interface

The ZNS SSD brings in two key benefits. First, it exposes controllable garbage collection to host applications, eliminating obtrusive I/O behaviors precipitated by device internal bookkeeping I/Os. This also alleviates write amplification and reduces flash over-provisioning. Second, it only allows sequential writes within a zone and thereby mitigates certain I/O interference observed in a conventional SSD. Both prior studies [3, 8, 16, 45] and our characterizations (§3) below demonstrate these points. However, existing ZNS SSDs have one significant drawback: **the zoned interface is static and inflexible**. After a zone is allocated and initialized, its maximum performance is fixed regardless of the underlying device capability, its I/O configurations cannot adapt to runtime workload characteristics, and cross-zone I/O interference yields unpredictable I/O executions.

First, the performance profile of a zone-sized storage partition hinges on physical zone placement and stripe configuration, which should align with application requirements. Despite significant benefits from the flexibility of the user-defined logical zone, application-managed zone configuration would sustain sub-optimal performance due to the lack of knowledge of other tenants sharing the device. In addition, it imposes another burden on application developers, as with OC SSDs.

Second, it is non-trivial to develop a complete application profile that captures every aspect of I/O execution characteristics, such as read/write block size and distribution, I/O concurrency, and command interleaving degree. The existing zoned interface fails to adapt to the changing workload behavior. Users have to over-provision the zone resources when configuring a zone based on the worst-case estimation.

In Figure 2, it is shown that the RocksDB over ZenFS [7] actively writes to only a fraction of the zones it maintains in the *active* state. This leads to inefficient utilization of valuable active resources in the ZNS SSD. Similarly, file systems like BtrFS [36] and F2FS [25] support ZNS SSDs but write user data to only one zone at a time, resulting in suboptimal utilization of the available active resources. This issue is fur-

ther exacerbated when the device has multiple namespaces serving different applications. In such cases, each application only utilizes a fraction of the available bandwidth, wasting valuable active resources in the ZNS SSD.

Third, a zone is not a completely performance-isolated domain, and co-located zones interact with each other in a non-deterministic fashion. Ideally, each tenant should receive a weighted share based on the consolidation degree. Specifically, its housing application should achieve its targeted performance when the SSD is under-utilized but receive a proportional degradation when the SSD is over-subscribed. However, unlike its predecessor OC SSD, ZNS SSDs manage zone allocation and wear-leveling internally with no strong isolation support and expose an opaque view to applications, yielding unpredictable performance interference and I/O execution unfairness. Such an issue could be mitigated in a conventional SSD where FTL and GC blend and distribute blocks across channels and dies uniformly regardless of the original command flow, ensuring the attainment of the maximum bandwidth and equal utilization of channel and die.

### 3 Performance Characterization of a ZNS SSD

This section characterizes a ZNS SSD with a focus on understanding why existing ZNS interfaces are static and inflexible. We then discuss the possibilities of addressing the problem.

#### 3.1 Experimental Setup

Device HW Parameters	Specification
Capacity	3,816 GB
Channels #	16 Channels
NAND Dies #	128 Dies
NAND Page Size	16 KB
NAND Channel B/W	~600 MB/s
Physical Zone Size	96 MB
Read B/W per Physical Zone	~200 MB/s
Write B/W per Physical Zone	~40 MB/s
Maximum Active Zones #	256

Table 1: The commodity ZNS SSD specification.

**ZNS SSD and testbed.** We use a commodity ZNS SSD for characterization. Table 1 presents its hardware details. It has 40,704 physical zones, where each 96MB-size zone consists of NAND erase blocks solely on a single die, and supports a maximum of 256 open zones simultaneously. We then configure various logical zones using such fine-granular units. We also prepare a conventional SSD with an equivalent architecture for a fair comparison. Our server has two 2.50GHz E5-2680v3 Xeon processors with 256GB DDR4 DRAM, and both SSDs are connected to  $\times 4$  PCIe Gen3 slots directly.

**Workloads and performance metrics.** We use the Fio benchmark tool [15] running on the SPDK framework [43] to generate synthetic workloads. We report both per-IO average/tail latency as well as achieved bandwidth. We add a thin layer to the SPDK to implement the logical zone concept and realize different zone configurations. Given the ZNS protocol, we regulate the write workloads to sequential accesses on a

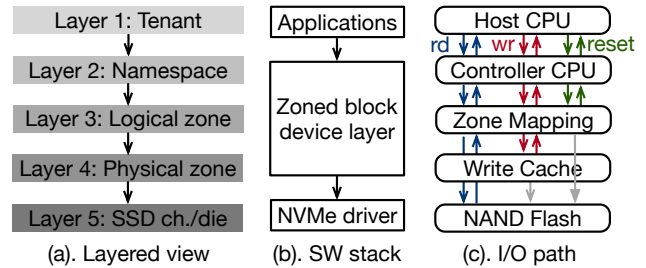


Figure 3: System model, SW stack, and I/O path of a multi-tenant ZNS SSD deployment. RD/WR=Read/Write. The write cache flushes data to the NAND flash asynchronously. Zone resets are completed after invalidating the mapping layer, where NAND blocks are erased lazily.

single logical zone in the following experiments, where read workloads are issuing random I/Os unless specified.

#### 3.2 System Model

We consider a typical system setup with a five-layered view to facilitate the understanding of a multi-tenant ZNS SSD deployment and dissect the I/O behavior (Figure 3-a). From the top-down perspective, the first layer contains a few co-located tenants, each running a storage application (e.g., blob store, F2FS, and RocksDB). Next, a tenant exclusively owns one or several namespaces based on the required capacity. A namespace provides independently configurable logical zones (layer 3), exposing a private logical block address space. By manipulating the logical zone setup, a namespace can be configured differently to meet the capacity and parallelism requirements. Within a logical zone, reads happen everywhere, while writes are only issued in an append-only manner. This is unique to a ZNS SSD and in significant contrast to a conventional SSD, which can be viewed as a fixed or statically configured SSD.

A logical zone comprises several physical zones (fourth layer). The number of physical zones per logical zone is typically fixed within a namespace. The logical-to-physical zone mapping can be arbitrary regardless of the request serving order and device occupancy. However, the logical zone must not share its physical zones with each other to conform with the ZNS protocol. At the bottom layer, a physical zone is placed on one channel/die following the device specification. The zoned block device (ZBD) layer (Figure 3-b) is the central component across the storage stack that abstracts away architectural details of a ZNS SSD. It provides three functionalities: (1) interacting with the application on namespace/logical zone management; (2) orchestrating the logical-to-physical zone mapping in consideration of the application requirement; (3) scheduling a sequence of I/O commands to maximize device utilization and avoid head-of-line blocking. Figure 3-c shows the IO path of read/write/reset requests. We carefully configure each layer when designing characterization experiments.

#### 3.3 Zone Striping

Since a logical zone is usually configured as an array of physical zones spatially, similar to RAID 0, one could apply the



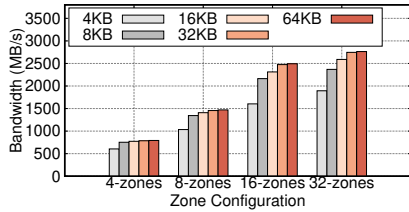


Figure 4: Read bandwidth varying the stripe size for different types of zones.

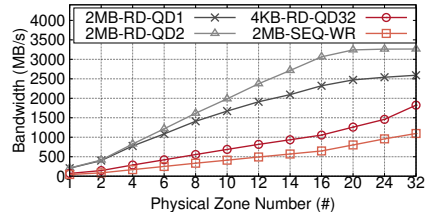


Figure 5: Read/Write bandwidth varying the number of physical zones.

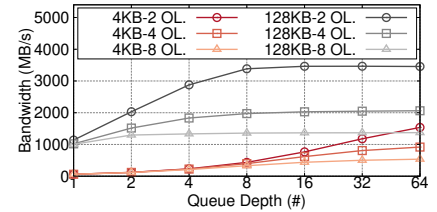


Figure 6: Read bandwidth under three channel overlapping (OL) allocations.

Stripe Size	Avg. Lat(us)	P99.9 Lat. (us)	B/W (MB/s)
4KB	64	76	59
8KB	71	84	108
16KB	88	103	175
32KB	163	269	190
64KB	314	619	198

Table 2: Read I/O average/P99.9 latency and bandwidth varying the stripe size on a physical zone.

striping technique to achieve higher throughput, especially for large-sized I/Os. Zone striping segments data blocks across multiple physical zones and access them concurrently. There are two configuration parameters: (1) *Stripe size* is the smallest data placement unit in a stripe, and (2) *Stripe width* defines the number of physical zones in an active state and controls the write bandwidth.

### 3.3.1 Basic Performance

When there are enough outstanding I/Os submitted to an SSD, unsurprisingly, the optimal striping efficiency is achieved when the stripe size matches the NAND operation unit (i.e., NAND page size). As shown in Table 2, the achieved per-die bandwidth increases slowly after the 16KB stripe size. In terms of latency, the access time reduction is non-linear for sizes smaller than a NAND page (16KB). When the I/O size is larger than 16KB, the average latency rises proportionally to the I/O unit because each request has to access the die multiple times sequentially. Next, we change the logical zone setup and see the efficiency of different stripe sizes. We use *N-zones* to refer to a logical zone configuration, where *N* is the number of physical zones in a striping. As shown in Figure 4, when issuing 2MB reads (which generates enough I/O to construct a full stripe I/O on each physical zone), for different zone configurations, the bandwidth over various stripe sizes shows a similar result with the single-die performance. On the other hand, a wider width that fully uses the stripe size ( $stripe\_size \times stripe\_width$ ) achieves higher bandwidth. For example, the 4KB stripe size in 8-zones achieves 37.3% higher read bandwidth than the 8KB stripe size in 4-zones. Note that the stripe size does not significantly affect the write performance as one can coalesce stripes on the same physical zone into a single device I/O and submit it at once. Instead, the stripe width determines the maximum write bandwidth.

### 3.3.2 Challenge #1: Application-agnostic Striping

When deciding the optimal stripe size and width, one should consider the application I/O profile dynamically, including

request type, size distribution, I/O size efficiency, and concurrency. However, the existing zoned interface lacks such support and hinges on users' domain knowledge during configuration. A large stripe may hurt performance if the size of sequential user I/O is smaller than the size of a full stripe. On the other hand, too small a stripe also hurts the I/O efficiency of the device; a 4KB stripe with an 8-zone or wider width significantly lags behind 8KB or larger stripes in Figure 4. A wide stripe width sustains high performance per logical zone. However, since the device has a limited amount of active resources, it will instead limit the maximum number of active logical zones and jeopardize application concurrency.

**Observation:** The use of logical zones with striping is beneficial for the application, but zone striping should be an adaptive configuration determined based on the total amount of active zones and application profiles. A ZNS SSD has to provide enough active logical zones to not only cope with application concurrency but also max out the device bandwidth by adjusting the stripe width dynamically. An ideal strip size can be the NAND page size, but it also has to be adjusted to the stripe width to provide a consistent full stripe size.

## 3.4 Zone Allocation and Placement

A ZNS SSD allocates physical zones across dies/channels, mainly taking access parallelism and wear-leveling into consideration. Upon an allocation request, the ZNS SSD traverses the die array following a certain order, and then selects the next available die to place each physical zone. Within a determined die, it chooses blocks with the least P/E cycles based on opaque wear-leveling policies.

### 3.4.1 Basic Performance

Zone allocation should be locality-aware and parallelism-aware. A larger-sized logical zone is expected to observe higher read/write bandwidth because it spreads physical zones across *different* channels and dies in a deterministic sequence and achieves more I/O parallelism. The maximum performance is obtained when I/Os access all channels and dies without blocking. We configure the stripe size to 16KB and increase the number of physical zones in a logical zone (*N*), then measure the I/O bandwidth of a single logical zone under four I/O profiles (Figure 5). The performance of 2MB reads with queue depths 1 and 2 (i.e., 2MB-RD-QD1/2MB-RD-QD2) keeps increasing until the number of physical zones approaches 20. But they max out for different reasons. The



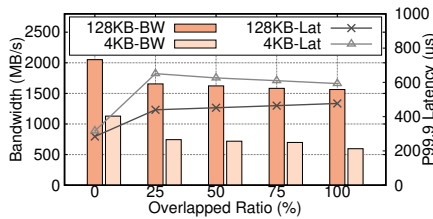


Figure 7: Bandwidth and tail latency varying with the die overlapping ratio.

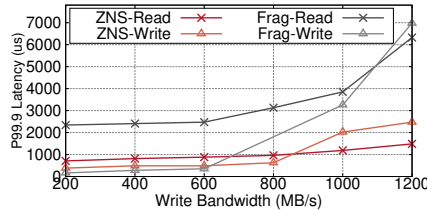


Figure 8: Read tail latency varying the write bandwidth (ZNS vs Conventional SSD)

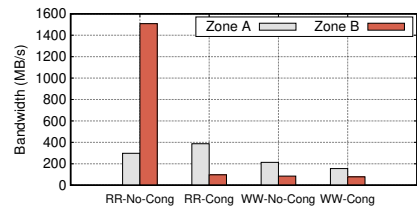


Figure 9: Bandwidth under RD-RD and WR-WR congestion due to the die-collision.

QD2 case is bounded by the PCIe bandwidth (i.e., four Gen3 lanes or 3.2GB/s), whilst the QD1 scenario is simply limited by the application as it cannot issue enough outstanding I/Os at that queue depth. In terms of 4KB random read with 32 queue depth and 2MB sequential write, they sustain 80MB/s read and 40MB/s program bandwidth per physical die, respectively, requiring much more physical zones ( $\sim 40$  and  $80$ ) to utilize the channel or PCIe bandwidth fully.

### 3.4.2 Challenge #2: Device-agnostic Placement

An ideal allocation process should expose all of the internal I/O parallelism of a ZNS SSD to a tenant. However, the existing mechanism is opaque to housed tenants, where the global allocation pointer picks the next available die without considering the application’s prior allocation history or how it interacts with other tenants. This causes unbalanced zone placement, hurts I/O parallelism, and jeopardizes performance. We find two types of inefficient placements:

- **Channel-overlapped placement:** Concurrent zone allocations might cause overlapped zone placements across channels, limiting the maximum channel parallelism. Similarly, synchronized allocation requests might prevent placement alignment, again limiting the aggregated bandwidth. Figure 6 presents 4KB and 128KB random read bandwidth when increasing the QD for three inferior placements, where 2/4/8 physical zones contend for the same channel in a 16-zone configuration. Physical zones stay across 16 different dies that limit the maximum bandwidth. The 2-overlapped allocation outperforms the other two (i.e., 4-overlapped/8-overlapped) by  $1.7\times/2.9\times$  and  $1.7\times/2.5\times$  for 4KB and 128KB cases, respectively.
- **Die-overlapped placement:** An intra-namespace die overlapped placement limits the bandwidth and can be even more detrimental because a die can only process one operation at a time. We configure such an experiment by placing physical zones in the same die and gradually increasing the overlapping ratio. Figure 7 reports the logical zone’s sustained bandwidth and tail latency under two I/O profiles. When no physical zones share the same die, it achieves 1,128MB/s and 2,051MB/s along with 317us and 284us p99.9 tail latency for the 4KB random read and 128KB sequential read cases, respectively. With full overlap, we observe 47.2%/23.8% bandwidth drop and 87.1%/28.0% tail latency increase. Such performance degradation hap-

pens even when the overlapping ratio is lower than 25%, because both types of I/Os suffer from the head-of-line blocking issue at the overlapped dies.

**Observation:** It is challenging to infer the zone’s physical location without knowing the device’s internal specification. One may run a profiling tool in the runtime to extract the relation among different zones [3]. However, it does not eliminate the imprinted overlap at the allocation time. To maximize the I/O parallelism, one could build a device abstraction layer that (1) relies on a general allocation model of the device; (2) maintains a shadow view of the underlying physical device; (3) profiles its placement balanced level across different physical channels and dies.

## 3.5 I/O Execution under ZNS SSDs

A ZNS SSD eradicates background GC I/Os, thereby removing one form of performance non-determinism. Within a logical zone, writes happen sequentially, but reads are issued arbitrarily. When reads are congested, one would observe latency spikes under die/channel contention. If considering cross-zone cases, either *intra* or *inter* namespace, interference would be more severe than a conventional SSD because ZNS SSDs impose no physical resource partitions, and per die/channel bandwidth is narrow.

### 3.5.1 Basic Performance

Irrespective of the NAND block layout of a logical zone, its I/O access latency highly correlates with achieved bandwidth because there are no device internal I/Os that consume bandwidth and are hidden from user applications. To demonstrate this, we prepare a conventional SSD having the same hardware as the ZNS SSD and compare two SSDs under the mixed read-write scenario. We configure a logical zone for the ZNS SSD that spreads across all the channels and dies (i.e., 128-zone configuration with 16KB stripe size) to match the conventional one. The fragmented conventional SSD is 70% filled and preconditioned with 128KB random writes. Then we run eight read threads—where each issues one 128KB read I/O to all the dies uniformly random—and one write thread that performs sequential write at a fixed rate. Figure 8 reports the read/write tail latency as we increase the write bandwidth. More writes on a ZNS SSD leave less bandwidth headroom for reads and cause the latency to increase. However, for the fragmented conventional SSD, the internal GC activities make even less bandwidth available to serve

reads due to write amplification. For example, when the write bandwidth is 1,000MB/s, the p99.9 read and write latency of the conventional SSD is  $4.3\times$  and  $2.8\times$  worse than the ZNS one. In terms of the read throughput, the conventional SSD shows  $1.1\times$  and  $1.6\times$  lower throughput than the ZNS SSD at the 200MB/s and 1,000MB/s write bandwidth, respectively.

### 3.5.2 Challenge #3: Tenant-agnostic Scheduling

Existing zoned interfaces of ZNS SSDs provide little performance isolation and fairness guarantees for the inter-zone case, regardless of deployed workloads. One cannot overlook the read interference on a die because (1) an arbitrary number of zones can collide on a die, (2) the bandwidth of a single die is poor, and hence the interference becomes severe even under a very low load on the device, and (3) it causes a severe head of line blocking problem and degrades the performance of the logical zone. Since there is no internal GC in the ZNS SSD, The I/O determinism [26] proposed for the conventional SSD does not apply as well. Similar to conventional SSDs, the write cache, shared among all NAND dies, is an indispensable component of the ZNS SSD, buffering incoming writes and flushing to the NAND dies in a batch. Host applications will observe prompt write I/O completions when they are absorbed by the cache but experience considerable latency spikes when the cache overflows. This has not been an intractable issue in conventional SSDs because the device firmware blends all incoming write I/Os and constructs a single large flow spanning entire NAND dies, maintaining the cache eviction rate to the maximum device bandwidth. However, in the ZNS SSD, a write I/O must be flushed out to the designated NAND die with an inadequate program bandwidth, even with zone striping. In this situation, a heavy writer exhausts the available cache capacity and severely disturbs other short flows.

We set up two readers performing 128KB read I/O in different profiles: (1) queue depth 8 with a two-zone configuration, and (2) queue depth 2 with an eight-zone configuration. Figure 9 shows the interference between two readers in a die-collision. The QD-8 reader easily obtains 97.2% of the total bandwidth of collision dies. Note that the interference and unfair bandwidth share also occurs in the conventional one, but only when the device bandwidth is fully saturated [23, 41]. We also demonstrate the write cache congestion in Figure 9. We first populate 15 logical zones with a stripe width of 8, and each physical zone is allocated to a dedicated die. The cumulative write bandwidth of 15 zones maxes out the PCIe bandwidth (3.2GB/s), and a single zone performs at  $\sim 213.3\text{MB/s}$ . In this case, a physical zone in the logical zone receives write at a lower rate than the maximum bandwidth ( $\sim 26.7\text{MB/s}$ ), and the write cache does not overflow. Then, we add one more writer with a narrow width of 2, which also runs on dedicated dies. Write I/Os towards the narrow zone are equally fetched by the device, but it soon consumes all available cache because of the scarce bandwidth ( $\sim 85\text{MB/s}$ ) of underlying physical zones. It degrades others' bandwidth by 27.3% or

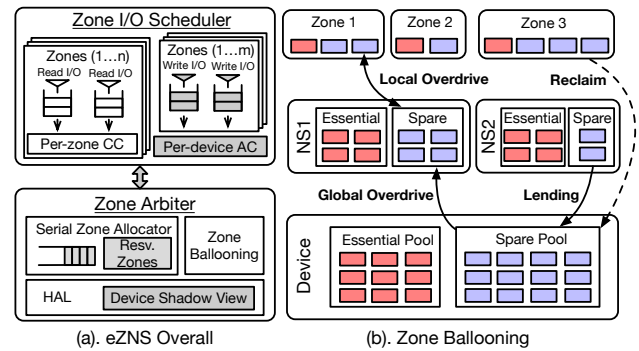


Figure 10: eZNS System Architecture.

155MB/s, and the device even fails to max out the PCIe bandwidth ( $\sim 2.4\text{GB/s}$ ).

**Observation:** When using ZNS SSDs in a multi-tenant scenario, one should first understand how different namespaces and logical zones share the channels and NAND dies of the underlying device, classify their relationships into competing and cooperative types, and employ a congestion avoidance scheme for the inter-zone scenario to achieve fairness. Since there are no device bookkeeping operations, I/O latencies represent the congestion level on colliding dies. In addition, write cache congestion needs to be addressed globally. Thus, a possible solution is to design (1) a global central arbiter that decides the bandwidth share among all active zones; (2) a per-zone I/O scheduler that orchestrates the read I/O submission based on the congestion level.

## 4 eZNS: Enabling an Adaptive Zoned NS

This section describes the design and implementation of eZNS that realizes a new and elastic zoned interface. We use the gathered insights from our characterization experiments and address the aforementioned issues.

### 4.1 eZNS Overview

eZNS stays atop the NVMe driver and provides raw block accesses. eZNS exposes the *v-zone* interface that offers runtime hardware adaptiveness, application elasticity, and tenant awareness. We carefully design eZNS and spread its functionalities across the control plane and data plane. As shown in Figure 10, it mainly consists of two components. The first is the zone arbiter that (1) maintains the device shadow view in a hardware abstraction layer (HAL) and provides the basis for other components, (2) performs serialized zone allocation avoiding overlapped placement, and (3) dynamically scales the zone hardware resources and I/O configurations via a harvesting mechanism. The second is a tenant-cognizant I/O scheduler, orchestrating read requests using a delay-based congestion control mechanism and regulating writes through a token-based admission control. In sum, eZNS addresses the three issues discussed in §3.

## 4.2 Hardware Contract and HAL

We develop eZNS based on the following hardware contract, which are met by recent ZNS SSDs with small zones: (1) a physical zone consists of one or more erasure blocks on a single die; (2) the maximum number of active physical zones is a multiple of the number of dies, and all dies hold the same number of active zones when they are fully populated (i.e., the ZNS SSD evenly distributes physical zones over dies); (3) the zone allocation mechanism follows the wear-leveling requirements, indicating that consecutive allocated zones will not overlap on a physical die until all the dies have been traversed. We need to caveat that the last contract may not always be followed in allocations if the device firmware enforces a specific policy other than round-robin across dies. However, considering the large number of chips and the wear-leveling constraint, such cases are rare. Our mechanism doesn't require being cognizant of the two-dimensional geometric physical view of SSD NAND dies and channels or maintaining an exact zone-die mapping.

eZNS maintains a shadow device view, exposing the approximate data locality for zone allocation and I/O scheduling. Our mechanism (or HAL layer) only hinges on three hardware parameters from device specifications. The first one is the *maximum number of active zones* (or MAR, maximum active resources). This is based on an observation that the MAR is generally in proportion to or a multiple of the number of physical dies on the SSD. One could estimate the number of active zones that a die could hold by deliberately controlling the zone allocation order in an offline calibration experiment (§3.4). The second parameter required is the *NAND page size* used for striping configuration. For example, 16KB is a de facto standard for most TLC NVMe drives and is well-known for system developers. The SSD shows the best efficiency when the stripe size is aligned with it (§3.3), and thereby, we choose the stripe size as a multiple or factor of the NAND page size that is closest to avoid inefficient stripe reads for sequential workloads. These two parameters reflect the device's capabilities. The third one is the *physical zone size*, deciding how a logical zone and strip groups are constructed. With such information, HAL provides a shadow view having a consistent MAR (e.g., 16) and the size of a zone (e.g., 2GB) regardless of the underlying device.

## 4.3 Serial Zone Allocator

eZNS develops a simple zone allocator that provides three guarantees: (1) it ensures that each stripe group comprises a list of consecutive and serial opened physical zones, following the firmware-enforced internal order; (2) there is no die collision within a stripe group; (3) across stripe groups, die collision could happen for writes only if available active physical zones are fully populated across all the dies. Given the above device model, the number of stripe groups colliding on a die is  $\frac{\text{Maximum \# of active zones}}{\text{Die \#}}$  at most. Channel collision would not be an issue because its bandwidth is usually higher

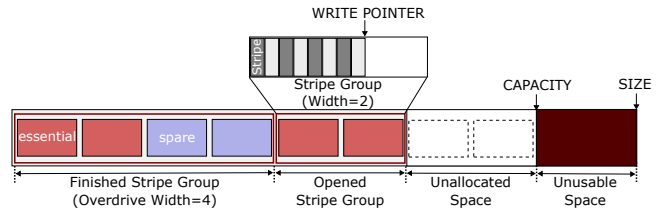


Figure 11: Example of eZNS *v-zone* structure.

than the aggregated program bandwidth across dies.

Our allocator works as follows. It has a per-device request queue, buffering OPEN commands (including implicit ones followed by writes) from all logical zones. Our allocator serves each logical zone request atomically. Since the completion of a zone OPEN command does not guarantee that the zone is actually allocated on a physical die, we implement a zone reservation mechanism during zone opens—flushing one data block that enforces binding a die to the zone. Writes complete immediately as the write cache of the device absorbs a single block even in high load. To expedite this process, we proactively maintain a certain amount of reserved zones in serial order and provision them to an upcoming stripe group. Upon completion of the allocation, we then update the allocation history and write it into a reserved persistent region (metadata block) following the block for reservation. Hence, we preclude interleaved allocations from concurrently opened logical zones to prevent channel-overlapped placement and facilitate allocation reordering to mitigate die overlaps (§3.4.2).

## 4.4 Zone Ballooning

*v-zone*, a specialized logical zone, can automatically scale its I/O striping configuration and hardware resources to match changing application requirements in a lightweight fashion. Figure 11 illustrates an example of a *v-zone* structure. Similar to a static logical zone, a *v-zone* contains a fixed number of physical zones. However, unlike a static logical zone, it divides physical zones into one or more stripe groups. When *v-zone* is first opened or reaches the end of a previous stripe group, it allocates a new stripe group. All physical zones in the previous stripe group must be finished when the write pointer reaches the end of the stripe group, allowing an active *v-zone* to take active resources for only one stripe group. The number of physical zones in a stripe group is determined at the time of allocation according to the *local overdrive* mechanism, which enables flexible zone striping. To comply with the standard zone interface, *v-zone* has a size that is a power of 2, and its capacity is the sum of user-available bytes in physical zones.

Similar to the virtualization memory ballooning technique [5, 39, 47], zone ballooning allows a *v-zone* to (1) expand its stripe width by leasing spares from others when other namespaces are under low active resource usage; (2) return them when it finishes the stripe group either by writing to the end of the stripe group or explicitly issuing FINISH/RESET commands from the application.



#### 4.4.1 Initial Resource Provisioning

eZNS divides all the available and opened physical zones on the ZNS SSD into two groups: *essential* and *spare*. The *essential* group contains a minimal number of active physical zones that can max out the SSD write bandwidth ( $N_{essential}$ ), whilst the rest belong to the *spare* group ( $N_{spare}$ ). Our initial resource allocation follows the equal bandwidth partition principle. We choose the write I/O bandwidth as the minimum guarantee because writing resources (or active physical zones) of a ZNS SSD are scarce. Assuming the number of namespaces that a ZNS SSD holds is  $N_{ns}$  and the maximum number of active  $v$ -zones per namespace is  $MAR_{logical}$ . A namespace takes  $\frac{N_{essential}}{N_{ns}}$  exclusive active physical zones; when a  $v$ -zone in the namespace opens a new stripe group, it receives  $\frac{N_{essential}}{N_{ns} \times MAR_{logical}}$  assured essential ones which is also the minimum stripe width. In terms of *spare* zones, similarly, eZNS equally distributes them to a namespace ( $\frac{N_{spare}}{N_{ns}}$ ) during initialization. Both a  $v$ -zone and a namespace will expand/shrink their capacity to adapt to workload demands.

#### 4.4.2 Local Overdrive: Zone Expanding

eZNS provisions available spares from the *spare* group of its namespace to boost its write I/O capability. We realize this via an internal *local overdrive* operation while opening a new stripe group. The mechanism works as follows. First, it estimates the resource usage of the namespace by analyzing its previously opened  $v$ -zones, quantified as the exponentially weighted moving average over the number of active  $v$ -zones ( $N_{ActiveZoneHistory}$ ). Second, it checks the remaining spares from the spare group ( $N_{RemainingSpare}$ ) and reaps additional spares based on  $\frac{N_{TotalSpare}}{N_{ActiveZoneHistory}}$ . Essentially, a  $v$ -zone will receive more (fewer) spares if it embodies writing activities but the namespace only opens fewer (more)  $v$ -zones. Third, the  $v$ -zone conflates the harvested spares with assured essential ones for it to open the new stripe group, and the stripe width is rounded down to the nearest power of two for efficient resource management. Note that the *local overdrive* operates in a serial and best-effort fashion. Lastly, eZNS sets the baseline stripe size to 32KB at the minimum width for the optimal I/O efficiency of the device. It then reduces the stripe size for an overdriven zone according to the stripe width, down to the minimum block size of the device. For example, if the width gets two times wider, the stripe size is reduced by half. We determine the range of stripe sizes to optimize the performance as aforementioned in §3.3. The reduced stripe size further contributes to the I/O scheduler ensuring fairness (§4.5).

#### 4.4.3 Global Overdrive: Namespace Expanding

Across the whole device, our zone ballooning mechanism further reallocates spares across namespaces based on their latest write activity. We realize this via another internal *global overdrive* operation—lend spares from the spare group to each other. Unlike *local overdrive*, global overdrive is triggered

based on the write intensity across the entire drive. Specifically, our arbiter monitors the past  $N_{essential}$  opened physical zones across all active namespaces, computes their zone utilization, and redistributes the remaining spares from inactive namespaces to active ones. In the current design, we determine an *inactive* namespace as a namespace that has no allocation history in the last  $N_{essential}$  physical zone allocations of the device, and lent spares are equally distributed across active namespaces. When an *inactive* namespace becomes active again, eZNS marks the leased spares as recall spares and leased namespaces release them to the global pool as soon as they FINISH/RESET the stripe group in  $v$ -zones. eZNS then returns them to the original namespace at the next *global overdrive* operation.

#### 4.4.4 Reclaim: Zone/Namespace Compaction

Generally, an overdriven  $v$ -zone after entering the FINISH state will return spare zones. Therefore, spare zones circulate as long as namespaces continue to write to  $v$ -zones. However, when a namespace overdrives  $v$ -zones, which becomes *inactive* without releasing them, the arbiter has to use a *reclaim* operation to take back the spares to prevent resource leakage. To ensure no slowdown on the performance path, we employ an asynchronous window-based monitoring scheme, where the arbiter bookkeeps the status of each *inactive* namespace and continuously counts how long its status is in the read-only state. If a namespace presents no write I/Os for a certain amount of time,  $T_{ReadOnly}$ , the arbiter triggers the reclaim procedure to proactively collect the spare zones. The execution cost of *reclaim* depends on the configuration within the opened stripe group. If there are committed writes on the zone, *reclaim* will trigger a zone compaction and perform a sequence of I/O reads/writes, i.e., finishing existing zones, opening a new stripe group with shrunk width, and copying data to the new one. Once the migration is done, the spare zones can be returned to the global spare pool.

The zone reclaiming indeed brings GC-like overheads back to the system. Thus, it is crucial that the system does not trigger the operation in normal conditions. In eZNS, zone reclaiming is only performed when namespaces have no write activity for two cycles of global overdrive. This is likely to happen infrequently, such as when an application undergoes a significant change in its running state. Moreover, reclaiming is triggered in a lazy fashion, executed in the background, and regulated by the scheduler to limit its performance impact. As a result, eZNS can avoid triggering zone reclaiming in normal conditions, maintaining high performance and efficiency.

### 4.5 Zone I/O Scheduler

eZNS mindfully orchestrates I/O reads/writes with the goal of providing equal read/write bandwidth shares among contending  $v$ -zones, maximizing the overall device utilization, and mitigating superfluous head-of-line blocking when different types of requests interleave. Our zone I/O scheduler com-



prises two components: congestion-avoiding read scheduler and cache-aware write admission control.

#### 4.5.1 Congestion-avoid Read Scheduler

Our design is based on the observations that (1) ZNS SSDs have no internal housekeeping operations; (2) write I/Os are sequential and synchronous. Hence, the read latency is stable and low until the die becomes congested, and it is thus possible to detect congestion directly via latency measurements.

eZNS introduces a hierarchical design that performs weighted round-robin scheduling firstly across active namespaces and then delay-based congestion control across each intra-namespace *v-zones*. By conforming to the NVMe architecture, we create per-namespace NVMe queue pairs and offload the round-robin scheduling to the device. Then, we employ a Swift-like [24] congestion control mechanism to decide the bandwidth allocation for each stripe group in the *v-zone*, where the delay is the device I/O command execution latency. As shown in Algorithm 1, during the congestion-free phase, upon a read I/O completion, we additively increase (AI) the congestion window until it approaches the maximum size (line 6). Since the congestion window (*cwnd*) is shared in the stripe group, when set to the stripe width, it indicates that there is one outstanding I/O per die in the sequential case. The SSD can max out its per-die bandwidth with a few outstanding I/Os. Thus, when the *cwnd* starts with the stripe width, it quickly ramps up to the device bandwidth capacity. Further, we limit the maximum congestion window (*cwnd*) to  $4 \times \text{strip\_width}$  to minimize the software overheads when handling excess concurrent I/Os and avoid a meaningless rapid growth of *cwnd* that would imperil the efficiency of the MD phase. When congestion happens, we reduce the congestion window multiplicatively (line 4), whose ratio depends on the latency degradation degree. All the physical zones within a stripe group share the same congestion status. It is reasonable because sequential read bandwidth will be capped by the most congested physical zone. Random reads usually will not trigger frequent *cwnd* decrements because the minimum window size is large enough to absorb them. Our congestion control works cooperatively with the reduced stripe size of the overdrive and ensures a fair share of bandwidth regardless of the width of the stripe group.

#### 4.5.2 Cache-aware Write Admission Control

Due to the non-linear write latency and the shared architecture, it is inappropriate to implement a local mechanism to mitigate the problem. Unlike the read congestion case, write congestion happens globally across all zones from all namespaces (§3.5). Therefore, eZNS monitors the global write latency and regulates writes using a token-based admission control scheme. We generate tokens periodically (ALG 1 lines 14–16) and admit write I/Os in a batch for each active *v-zone* to ensure overflow rarely happens. This requires a latency monitor to analyze the write cache eviction activity (ALG 1 lines 8–12). Here, we profile the block admission rate (defined as

---

#### Algorithm 1 Zone I/O Scheduler

---

```

1: procedure READ COMPLETION()
2:   lat_thresh  $\leftarrow$  500us
3:   if io_lat > lat_thresh then
4:     cwnd = max(1, cwnd  $\times$   $\frac{\text{lat\_thresh}}{2 \times \text{io\_lat}}$ )
5:   else  $\triangleright \alpha = \text{additive factor}$ 
6:     cwnd = min(stripe_width  $\times$  4, cwnd +  $\alpha \times \frac{\text{io\_count}}{\text{cwnd}}$ )
7: procedure WRITE LATENCY MONITOR()
8:   On t every 10ms
9:   total_lat =  $\sum_{\text{active\_zone}} \text{per\_block\_lat}$ 
10:  total_ios =  $\sum_{\text{active\_zone}} \text{num\_ios}$ 
11:  avg_lat(t) =  $\frac{\text{total\_lat}}{\text{total\_ios}}$ 
12:  block_admission_rate =  $\frac{\text{avg\_lat}(t-1) + \text{avg\_lat}(t)}{2}$ 
13: procedure WRITE TOKEN GENERATOR()
14:   On every 1ms
15:   for pending write zones do
16:     token +=  $\frac{\text{now} - \text{last\_refill}}{\text{block\_admission\_rate}} \times \text{stripe\_width}$ 

```

---

the minimum delay between two consecutive write blocks) and adjust the token generation rate based on its normalized average latency. This is based on an empirical observation that the latency of the write projects its capacity share in the write cache. Hence, we equalize the latency for all write zones and calculate available tokens using the average value. Additionally, we update the available tokens based on the elapsed time from the last token refill upon a write submission. By doing so, we expect that writes are self-clocked in the congestion-less condition.

Note that (1) when read and write I/Os mix on a physical die, the total aggregate bandwidth will drop due to the NAND interference effect. However, our read scheduler and write admission control require little coordination because both modules only use the latency (gradient) as a signal to infer the current bandwidth capacity; (2) we coalesce stripes for the same physical zone within a user I/O and submit one write I/O to the device in a batch, and thus, a small stripe size does not degrade the write bandwidth.

## 5 Evaluation

We add a thin layer in the SPDK framework [43] to implement eZNS and realize the *v-zone* concept. The primary reason for choosing the SPDK approach was its ease of implementation and integration into the software stack of a storage server accessible by remote clients. Moreover, the SPDK-based design can also be used in a local system to serve virtual machines through the SPDK vhost extension. This approach allows the storage server to provide efficient and high-performance I/O operations, while remaining compatible with existing software stacks. We use the same test environment as in §3.1. Non-SPDK applications require a standard ZNS block device exposed via the kernel NVMe driver; thus, we set up eZNS as a disaggregated storage device over RDMA (NVMe-over-RDMA) and connect to it using the kernel NVMe driver.

**Micro-benchmarks:** We use FIO [15] to generate syn-

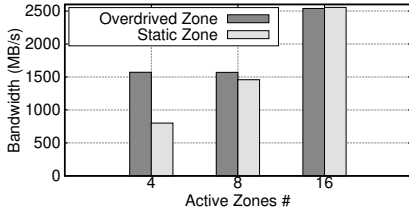


Figure 12: B/W comparison between an overdriven and three statically configured zones.

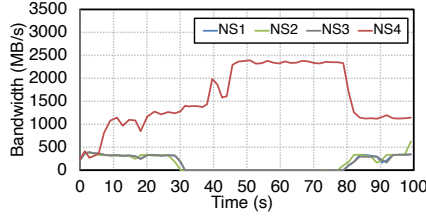


Figure 13: Performance variation of four namespaces with global overdrive under 100s.

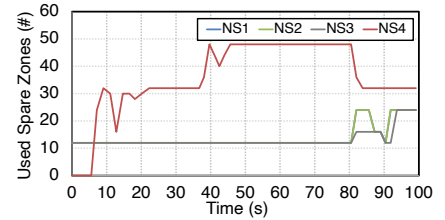
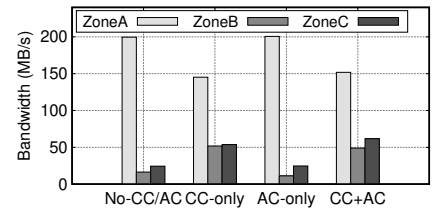
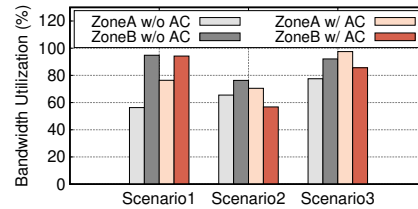
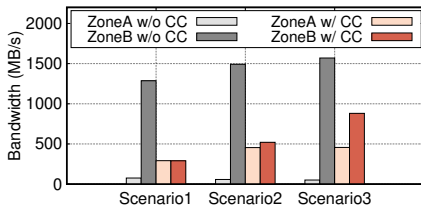


Figure 14: The number of used spare zones of four namespaces under 100s.



(a) Read-Read Fairness. (128KB Read. Zone A with QD-1, and Zone B with QD-32) (b) Write-Write Fairness. (Zone A for regular writers, and Zone B for the busy writer) (c) Read-Write Fairness. (Zone A for readers, Zone B for the busy writer, and Zone C for regular writers)

Figure 15: Efficiency of eZNS on handling read-read, write-write, and read-write congestion. (CC=Congestion Control, AC=Admission Control)

thetic workloads and allocate a separate thread for each worker when the workload writes to multiple namespaces or zones. For read workloads, we first precondition the namespace by performing sequential writes for the entire range of read I/O. Additionally, we perform a pre-calibration step to determine the die allocations in case the evaluation requires a die-level collision.

**Ported Applications:** We use RocksDB as a real-world ZNS application, to evaluate the performance of eZNS We run RocksDB over ZenFS [7] to enable the ZNS support. As eZNS complies with the standard NVMe ZNS specification, no modification is required for the application and ZenFS. We initialize the DB instance with 500M entities of 20-byte keys and 1,000-byte values.

**Default  $v$ -zone Configuration:** By default, eZNS creates four namespaces (NS1–4), each of which is allocated 32 essential and 32 spare resources. Since each namespace provides a maximum of 16 active zones, the minimum stripe width for  $v$ -zone is 2 with a stripe size of 32KB. However, eZNS can overdrive the width up to 16 with a stripe size of 4KB. For a fair comparison, we prepare a static logical zone configured with stripe width and size of 4 and 16KB, respectively; hence, it also accesses full device capability when the application populates enough active logical zones. Both a  $v$ -zone and a static logical zone comprise 16 physical zones. Different configurations are used for single-tenant evaluation (single namespace) as specified in Section 5.3.

## 5.1 Zone Ballooning

We demonstrate the efficiency of zone ballooning when handling large writes (i.e., 512KB I/O with a queue depth of one). First, within a namespace, we compare the performance be-

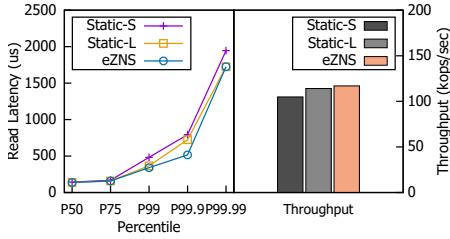
tween a  $v$ -zone and a static logical zone, where the number of writers is configured to 4, 8, and 16, respectively. Each writer submits a write I/O to different zones. Our *local overdrive* operation can reap more spare zones and lead to better throughput. As shown in Figure 12, the  $v$ -zone outperforms the static one by  $2.0\times$  under the 4-writer case as 4 static logical zones enable only 16 physical zones while 4  $v$ -zone overdrive the width to 8 and expand to 32 physical zones. In the 8-writer and 16-writer cases,  $v$ -zone reduces the overdrive width accordingly and utilizes the same number of physical zones (32 and 64, respectively) with the static logical zone.

To evaluate eZNS’s adaptiveness under dynamic workloads, we set up overdriven zones from different namespaces. The first three namespaces (NS1, NS2, and NS3) run two writers, while the fourth namespace (NS4) runs eight. NS1, NS2, and NS3 stop issuing writes at  $t=30s$  and resume the writing activity at  $t=80s$ . We measure the throughput and spare zone usage of four zones for a 100s profiling window (Figures 13 and 14). When the other three zones become idle, the  $v$ -zone from NS4 takes up to  $3\times$  more spare zones from other namespaces using the *global overdrive* primitive and maxes out its write bandwidth ( $\sim 2.3GB/s$ ). It can then quickly release the harvest zones when other zones start issuing writes again.

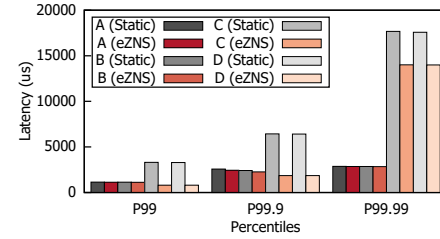
## 5.2 Zone I/O Fairness

We evaluate our I/O scheduler in various synthetic congestion scenarios by placing competing zones in the same physical die group. We compare the performance of all co-located zones when enabling and disabling our mechanism. The zone ballooning mechanism is turned off for all cases. We report per-thread bandwidth in Figure 15.

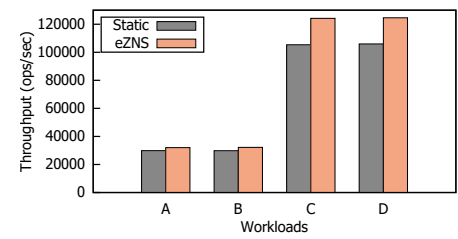
**Read-Read Fairness.** We run a sequential read of 128KB



**Figure 16: readwhilewriting workload on a single tenant configurations. Static has stripe width of 16. (S: 4KB stripe, L: 16KB stripe)**



**Figure 17: Latency of db\_bench workloads (2 overwrite, 2 randomread) on different namespaces over eZNS and static zone.**



**Figure 18: Throughput of db\_bench workloads (2 overwrite, 2 randomread) on different namespaces over eZNS and static zone.**

I/O size at two types of zones on co-located dies. To equally load the physical dies, we populate more threads for lower-width zones. For example, a zone with a width of 2 runs four threads on each stripe group, while a zone with a width 8 has only one thread. As shown in Figure 15-a, in scenario 1, when disabling our congestion control mechanism, Zone A (configured with stripe width 2 and stripe size 32KB, QD-1) and Zone B (configured with stripe width 8 and stripe size 8KB, QD-32), even holding the same sized full stripe, achieve 76MB/s and 1287MB/s, respectively. This is because the zone with the higher QD dominates on the competing die. Our scheme effectively controls the per-zone window size and ensures that each zone submits the same amount of outstanding bytes. Hence, both Zone A and Zone B sustain 290MB/s. In scenarios 2 and 3, we change the Zone A stripe configuration to <stripe width 4, stripe size 16KB, QD-1> and <stripe width 8, stripe size 8KB, QD-1>, and observe similar behavior when turning off the read congestion logic. In scenario 3, the congestion level on the die gets lowered as Zone A only submits one 128KB I/O (which was 4 and 2 in scenarios 1 and 2, respectively). Hence the read latency also becomes below the threshold, and the I/O scheduler chooses to max out the bandwidth.

**Write-Write Fairness.** We carefully create different write congestion scenarios and see how our admission control operates. The workload used is a sequential write of 512KB size. In the first scenario, we co-locate 16 regular write zones (Zone A, where each has a striping width of 8 with 8KB stripe size and submits write I/Os at 5ms intervals, sustaining 95MB/s maximum throughput) with a busy writer (Zone B, that has width 2 and 32KB stripe size, submits I/O without interval delays, achieving 85MB/s at most). Figure 15-b reports the bandwidth utilization of one regular zone (Zone A) and the busy writer (Zone B). Our admission control mechanism limits the write issuing rate of Zone B and gives more room at the write cache to the regular zone (Zone A), leading to 35.7% bandwidth improvement per thread. Next, we set up a highly-congested case by changing 16 regular zones to busy writers (scenario 2). As described in §4.5.2, our scheme equally distributes the write bandwidth share across competing zones, and Zone B receives 56.8% of the total bandwidth of 2 physical zones. The last scenario is a collision-less one at the die level where we eliminate the overlapping region among all the

write zones by populating active physical zones lesser than the number of dies. Similarly, when enabling the admission control, the bandwidth allocated for Zone B slightly decreases ( $\sim 7.2\%$ ) to avoid cache congestion, and the overall device bandwidth is increased by 24.7%.

**Read-Write Fairness.** We examine how our congestion control mechanism coordinates with the admission control when handling read/write mixed workloads. In this experiment, we set up three types of zones: (1)  $\times 16$  regular readers (Zone A), where each has a striping width of 2 and 32KB stripe size, performing 128KB random read at queue depth 32, across all physical dies; (2) 1 busy writer (Zone B), whose striping width is 2 with 32KB stripe size; (3)  $\times 16$  regular writers (Zone C), which has a striping width of 8 and 32KB stripe size each, submitting I/Os under 5ms interval. Both B and C issue 512KB large writes. Figure 15-c reports their per-thread bandwidth. When disabling our scheduler, each reader achieves 199.6MB/s but writes are jeopardized significantly, where Zone B and Zone C can only achieve 19.3% and 27.3% of their maximum bandwidth. As we gradually turn on our mechanisms, the congestion control shrinks the window size such that more bandwidth is allocated to the writes. Further, the admission control then equally partitions bandwidth among competing writing zones. As shown in the CC+AC case, zone A, B, and C can sustain 71.6%, 57.5%, and 70.1% of their maximum bandwidth capacity, respectively.

### 5.3 Application: RocksDB

To evaluate eZNS in a real-world scenario, we use RocksDB [35] over the ZenFS storage backend. In addition to the built-in utility in the RocksDB *db\_bench* tool, we port YCSB workload generators [4] for the mixed workload evaluation.

**Single-tenant performance.** First, we evaluate the performance of a single tenant using the *readwhilewriting* profile of the *db\_bench*, which runs one writer and multiple readers. This workload profile demonstrates a read/write mixed scenario. In the case of a single-tenant configuration, eZNS creates a single namespace on the device and allocates 128 essential and 128 spare resources to it. Since only two stripe widths, 8 and 16, are possible in this configuration, eZNS sets the stripe size to 16KB for the width of 8 to avoid the namespace running only on large stripe sizes. We compare the performance of eZNS over two static configurations, both

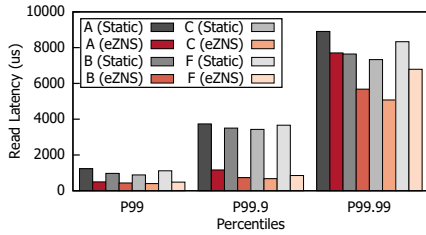


Figure 19: Read latency of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

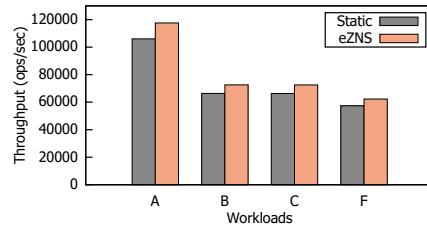


Figure 20: Throughput of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

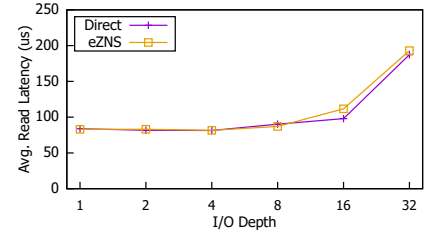


Figure 21: Comparison of Avg. Read Latency for 4KB I/Os at various depths between the host-managed zone access and eZNS.

with a stripe width of 16, but with different stripe sizes of 4KB and 16KB. Since there is only one namespace on the device, eZNS always overdrives *v-zones* to the width of 16, which is identical to the static configurations. Therefore, both the static namespace and eZNS can exploit all available bandwidth on the device. However, the I/O scheduler of eZNS helps mitigate interferences between zones and improves overall application performance. Figure 16 shows that eZNS improves the P99.9 and P99.99 read latency by 28.7% and 11.3% over the static configurations with a stripe size of 16KB and 4KB, respectively. Additionally, eZNS also improves the throughput by 11.5% and 2.5% with a stripe size of 4KB and 16KB.

**Multi-tenant Performance.** Next, we set up instances of *db\_bench* on four namespaces (A, B, C, and D), each with a different workload profile. A and B perform the *overwrite* profile, while C and D execute *randomread* concurrently. We run the benchmark for 1,800sec and report the latency and the throughput. Figure 17 shows that our I/O scheduler significantly reduces P99.9 and P99.99 read (C/D) latency by 71.1% and 20.5%, respectively. In terms of throughput, eZNS improves write (A/B) and read (C/D) throughput by 7.5% and 17.7%, respectively. Furthermore, while the read latency and throughput are improved, the write latency is either maintained at the same level or decreased compared to the static configuration because eZNS moves the spare bandwidth from read-only namespaces (C/D) to write-heavy ones (A/B).

**Mixed YCSB Workloads.** YCSB [14] is widely used to benchmark realistic workloads. In our experiments, we run YCSB workload profiles A, B, C, and F on each of the six namespaces. We exclude YCSB workload profiles D and E because they increase the number of entities in the DB instance during the benchmark. As YCSB-C (read-only) does not submit any write I/Os during the benchmark, eZNS triggers *global override* and rebalances the bandwidth to the write-most namespaces (A and F). Figure 19 shows that the I/O scheduler improves the P99.9 read latency of read-intensive workloads (YCSB B and C) and also the read-modify-write one (YCSB F) by 79.1%, 80.3%, and 76.8%, respectively. The throughput improvement from global override is up to 10.9% for the write-most workload A in Figure 20.

## 5.4 Overhead analysis

**End-to-end read latency overhead.** Since eZNS serves as an orchestration layer between the physical ZNS device and the NVMe-over-Fabrics target, there may be some overhead when the I/O load is very low. To measure this overhead, we conducted a quantitative analysis using 4KB random read I/Os and compared it with host-managed zone access, where the host directly accesses the physical device without eZNS. Figure 21 demonstrates that eZNS does not add a noticeable latency overhead for I/O depths up to 8. As the I/O depth goes over 16, up to 14.0% overhead is observed due to the I/O scheduler delaying the I/O submission. However, the scheduler provides significant advantages in real-world scenarios as shown in previous experiments.

**Memory footprint.** eZNS relies on in-memory data structures for managing *v-zone* metadata, including the logical-to-physical mapping and scheduling statistics. Additionally, it maintains a copy of the physical zone information to reduce unnecessary queries to the device, enabling faster zone allocation and deallocation. In our current implementation, the size of *v-zone* metadata is less than 1KB, and the size of physical zone information is smaller than 64 bytes. For our testbed SSD with four namespaces, each with 1TB of capacity, *v-zone* metadata and physical zone information require 2MB and 2.5MB of memory, respectively. Compared to the memory requirements of the page-mapping in conventional SSDs, the memory usage of eZNS is negligible.

## 6 Related Work

**Early ZNS Exploration.** Researchers have made initial efforts to understand the ZNS interface and integrate it into the host storage stack. Theano Stavrinou *et al.* [44] argue for a shift in research to the zone interface and discuss future directions (e.g., applying application-level information for zone management and I/O scheduling). Hojin Shin *et al.* [42] develop a performance analysis tool for a ZNS SSD and profile its parallelism, isolation, and predictability properties. Compared with our study, they didn't investigate the underlying device's internal mechanisms when realizing the zoned namespace interface and, thereby, are unable to correlate the observed performance with the ZNS SSD characteristics. ZNS+ [16] enhances the existing interface with two



new architectural primitives to optimize LFS file systems. With such support, the authors then propose copy-back-aware block allocation and hybrid segment recycling techniques. Hanyeoreum Bae *et al.* [3] prioritize I/O requests for less congested zones using an interference map, whilst updates incur significant overheads. Although revising the ZNS interface and exposing the physical allocation of zones could potentially eliminate this overhead, it may not be feasible for existing devices due to vendors' resistance to disclosing internal architecture and policies. eZNS uses a delay to determine congestion and doesn't require an allocation map. Furthermore, eZNS addresses such as read and write differences, zone striping, and bandwidth provisioning issues that were not discussed in their work. Minwoo Im *et al.* [18] improved ZenFS on small-zone SSDs by introducing read/write parallelism with a multi-threaded I/O engine and lifetime-based zone management at the application level. However, it requires adjusting the RocksDB parameters to match the device capability instead of the workload-optimized parameters. This can increase the complexity of parameter configuration, resulting in sub-optimal settings for the workload. eZNS maximizes parallelism within the thin layer, regardless of the underlying device and the application profile. It exploits the device's parallel I/O processing capability that can be executed on a single thread.

**Addressing Inefficiencies of Conventional SSDs.** Early SSD researches [2, 11, 17, 31] focused on internal parallelism and tradeoffs between concurrency, locality, bandwidth, capacity, performance, and lifetime. Modern SSDs handle random write patterns with page mapping FTL, write-cache, and superbblock concepts [49] that group blocks together. It benefits from high parallelism that transforms writes into sequential NAND programming. However, multi-tenancy workloads cause interference and high write amplification factor (WAF). ZNS SSDs eliminate garbage collection and fix WAF to one, but require careful parallelism management across zones to avoid degraded device utilization. In addition, future QLC-based ZNS SSDs may have fewer active zones due to a multi-pass programming algorithm [21]. eZNS addresses these challenges by adjusting the parallelism of each logical zone based on the number of namespace flows, providing fully dynamic parallelism and maximizing device capability while presenting an identical logical view to applications.

IODA [26] is an I/O deterministic flash array that uses the I/O determinism feature and exploits data redundancy for a strong latency predictability contract. SSDs can fail an I/O to allow predictable I/Os through proactive data reconstruction. We target the ZNS SSD, where there are no random I/Os, and GCs are user-controlled. This opens up a different design space. Although techniques addressing GC-related interference are not beneficial to GC-free ZNS SSDs, others such as Engurance Group(EG) and NVM Set can be useful to ensure physically-isolated zone allocation. eZNS can take advantage of the geometry hints via EG (or even finer-grained NVM

Sets). Unfortunately, there is no currently-available SSD that supports both ZNS and EG, but it will be an interesting direction for future work.

**Open-Channel SSDs.** These drives have no mapping layer in the controller and directly expose a set of physically contiguous blocks to applications, and leave the data placement/wear-leveling responsibilities to the host. Researchers have built several domain-specific solutions using them. For example, SDF [30] employs a hardware-software co-designed approach that exposes flash channel details and delegates I/O control-plane and data-plane tasks to host applications. LOCS [48] further improves the throughput of an LSM-tree-based KV store by optimizing the scheduling and dispatching policies, considering the characteristics of access patterns of the LevelDB. RAIL [27] designs a horizontal hot-cold separation mechanism and divides dies into two groups, where user and GC writes are scheduled to different dies, and the hot/cold ratio is dynamically adjusted based on runtime monitoring. By having full control over the device, one can implement a deterministic *v-zone* using eZNS. Despite the potential architecture, it imposes too many responsibilities on the software handling tasks that are offloadable to the device with no cost, for example, wear-leveling, physical zone-to-die mapping, etc. Another challenge arises when the system consists of heterogeneous devices resulting in the overhead of managing different H/W architectures (NAND chip capacity, channel/die configuration, etc.).

**eZNS as a firmware.** One may implement eZNS solely in the SSD using the controller and firmware. This approach can exploit internal knowledge such as NAND specification, Channel/Die structure, queue length on a die, etc. Thus, it may control the interference better and outperform the software-based implementation. However, completing eZNS in one device is not future-proof, given the disaggregated systems architecture in data centers. The software-based solution can build an eZNS-based system spanning multiple devices enabling elastic capacity scaling, load-aware allocation, high availability, and more.

## 7 Conclusion

This paper presents an in-depth study on understanding the characteristics of a commodity ZNS SSD. Then, we propose eZNS, realizing an elastic zoned view via *v-zone*, providing a flexible zone scaling interface transparent to the application that maxes out the device capability, and ensuring a fair bandwidth share between zones. We demonstrate significant performance and fairness improvements using eZNS over various scenarios.

## Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Mark Silberstein. This work was supported in part by NSF grant CNS-2212193 and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

## References

- [1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, 2008.
- [2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 57. Boston, USA, 2008.
- [3] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What You Can't Forget: Exploiting Parallelism for Zoned Namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022.
- [4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. {SILK}: Preventing latency spikes in {Log-Structured} merge {Key-Value} stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.
- [5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.
- [6] Matias Bjørling. From open-channel ssds to zoned namespaces. In *Linux Storage and Filesystems Conference (Vault 19)*, volume 1, 2019.
- [7] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.
- [8] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
- [9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. {LightNVM}: The linux {Open-Channel}{SSD} subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, 2017.
- [10] Feng Chen, Binbing Hou, and Rubao Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Trans. Storage*, may 2016.
- [11] Feng Chen, Binbing Hou, and Rubao Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage (TOS)*, 12(3):1–39, 2016.
- [12] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.
- [13] Feng Chen, Tian Luo, and Xiaodong Zhang. {CAFTL}: A {Content-Aware} flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.
- [14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- [15] Flexible I/O Tester (FIO). <https://github.com/axboe/fio>, 2022.
- [16] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.
- [17] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2012.
- [18] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, Middleware Industrial Track '22, page 15–21, New York, NY, USA, 2022. Association for Computing Machinery.
- [19] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2013.
- [20] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superbblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.

- [21] Ali Khakifirooz, Sriram Balasubrahmanyam, Richard Fastow, Kristopher H Gaewsky, Chang Wan Ha, Rezaul Haque, Owen W Jungroth, Steven Law, Aliasgar S Madraswala, Binh Ngo, et al. 30.2 a 1tb 4b/cell 144-tier floating-gate 3d-nand flash memory with 40mb/s program throughput and 13.8 gb/mm<sup>2</sup> bit density. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 424–426. IEEE, 2021.
- [22] Moosung Kim, Sung Won Yun, Jungjune Park, Hyun Kook Park, Jungyu Lee, Yeong Seon Kim, Dae-hoon Na, Sara Choi, Youngsun Song, Jonghoon Lee, Hyunjun Yoon, Kangbin Lee, Byunghoon Jeong, Sanglok Kim, Junhong Park, Cheon An Lee, Jaeyun Lee, Jisang Lee, Jin Young Chun, Joonsuc Jang, Younghwi Yang, Seung Hyun Moon, Myunghoon Choi, Wontae Kim, Jungsoo Kim, Seokmin Yoon, Pansuk Kwak, Myunghun Lee, Raehyun Song, Sunghoon Kim, Chiweon Yoon, Dongku Kang, Jin-Yub Lee, and Jaihyuk Song. A 1tb 3b/cell 8th-generation 3d-nand flash memory with 164mb/s write throughput and a 2.4gb/s interface. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 136–137, 2022.
- [23] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash= local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.
- [24] Gautam Kumar, Nandita Dukkkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.
- [25] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.
- [26] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.
- [27] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Trans. Storage*, 18(1), 2022.
- [28] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.
- [29] The NVMe Base Specification. <https://nvmexpress.org/developers/nvme-specification/>, 2022.
- [30] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
- [31] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, 2008.
- [32] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.
- [33] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Saša Tomić, Nikolaos Papandreou, Thomas Parnell, Haralampus Pozidis, Aaron Fry, and Tim Fisher. Management of next-generation nand flash to achieve enterprise-level endurance and latency targets. *ACM Transactions on Storage (TOS)*, 14(4):1–25, 2018.
- [34] Radian Memory System RMS ZNS SSDs. [https://www.radianmemory.com/zoned\\_namespaces/](https://www.radianmemory.com/zoned_namespaces/), 2022.
- [35] RocksDB. <http://rocksdb.org/>, 2022.
- [36] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.
- [37] Samsung PM1731a ZNS SSDs. <https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan>, 2022.
- [38] Samsung PM1731a Review from STH. <https://www.servethehome.com/samsung-pm1731a-ssd-with-zns-support/>, 2022.
- [39] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, volume 2, pages 313–319, 2006.
- [40] The SCSI Protocol. [https://en.wikipedia.org/wiki/SCSI#cite\\_note-1](https://en.wikipedia.org/wiki/SCSI#cite_note-1), 2022.

- [41] Kai Shen and Stan Park. {FlashFQ}: A fair queueing {I/O} scheduler for {Flash-Based}{SSDs}. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.
- [42] Hojin Shin, Myoungsoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring Performance Characteristics of ZNS SSDs: Observation and Implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2020.
- [43] The Storage Performance Development Kit (SPDK). <https://spdk.io>, 2022.
- [44] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don’t Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021.
- [45] Nick Tehrani and Animesh Trivedi. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices, 2022.
- [46] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 35–40. IEEE, 2011.
- [47] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, dec 2003.
- [48] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.
- [49] Shunzhuo Wang, Fei Wu, Chengmo Yang, Jiaona Zhou, Changsheng Xie, and Jiguang Wan. Was: Wear aware superblock management for prolonging ssd lifetime. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.
- [50] Western Digital Ultrastar ZNS SSDs. <https://www.westerndigital.com/solutions/zns>, 2022.
- [51] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don’t Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.
- [52] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing*, pages 66–73. IEEE, 2014.
- [53] Yiyang Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.
- [54] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of {SSDs} under power fault. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 271–284, 2013.







# SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory

Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang  
*The Chinese University of Hong Kong*

## Abstract

With the merits of high density, non-volatility, and DRAM-scale latency/bandwidth, persistent memory (PM) brings hope to high-performance storage systems, in which hashing-based index structures receive great attention owing to the efficient query performance. Though lots of efforts have been made to rethink the hashing schemes for PM in recent years, nevertheless, based on our investigation, none of them can hit performance scalability, efficiency, and predictability with one stone, seriously limiting their practicality to time-sensitive or latency-critical applications. To this end, this paper presents SEPH, a Scalable, Efficient, and Predictable Hashing for PM. SEPH paves a new direction to build the hash table by introducing the novel Level Segment (LS) structure, a key to breaking the dilemma between efficiency and predictability standing in front of the existing hashing schemes for PM. With the LS-based hash table structure, SEPH further enables a low-overhead split to greatly suppress the resizing-incurred unpredictability, and develops a semi lock-free concurrency control that requires a nearly-minimal amount of writes to handle an item insertion for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency. Compared to state-of-the-art hashing schemes, SEPH demonstrates higher efficiency (up to 15.4× higher throughput), better scalability (performance scales up to 48 threads), and more reliable predictability (improving the tail latency by up to 19.3×).

## 1 Introduction

Persistent memory (PM) offers storage systems the potentials of large capacity, low latency, high throughput, and instant recovery [8, 48]. The first commercial product of PM, i.e., Intel® Optane™ DC Persistent Memory Module (DCPMM) [3], is currently available on the market. As shown in Table 1, compared with DRAM, Intel® Optane™ DCPMM delivers similar write latency yet has about 2× sequential read latency and 3× sequential read latency [20, 21, 45]; besides, the read and write bandwidths of Intel® Optane™ DCPMM achieve nearly 1/3

Table 1: Performance Comparison between DRAM and PM (i.e., Intel® Optane™ DCPMM 100 Series) [45].

	DRAM	PM	PM/DRAM
Latency of Seq. Read (ns)	81	169	208.64%
Latency of Ran. Read (ns)	101	305	301.98%
Latency of Write (ns)	57	62	108.77%
Bandwidth of Read (GB/s)	105.6	37.6	35.61%
Bandwidth of Write (GB/s)	76.8	12.5	16.28%

and 1/6 of those of DRAM [20, 21, 26]. When compared with SSD, Intel® Optane™ DCPMM is even much more superior in every of these performance metrics [45]. Together with the maximal 512 GB capacity for a single module, Intel® Optane™ DCPMM is especially attractive to in-memory applications [43, 45].

Index structure is a vital component for high-performance storage systems to offer efficient queries. To rapidly deploy the well-developed indexes on PM, RECIPE [26] presents a principled approach to convert concurrent DRAM indexes, including tree-based and hashing-based indexes, into crash-consistent indexes for PM. However, to better unleash the full potentials of PM, more researches focus on developing carefully-tailored indexes for PM. For example, a series of researches develops tree-based indexes for PM especially, like NV-tree [46], FAST&FAIR [19], wB+-Tree [9], LB+-Trees [31], WORT [25], BzTree [5], and ROART [35]. However, the search operation of tree-based indexes usually performs in the complexity of  $O(\log N)$ , where  $N$  is the size of data structure, because of the hierarchical structure of trees.

By contrast, hashing-based indexes can provide constant-scale query time complexity due to the flat structures, so they are widely adopted by in-memory systems [18, 23, 29, 47]. Hashing indexes can be generally categorized into two classes: *static* and *dynamic*. Static hashing must estimate and allocate sufficient space in advance, but it suffers from hash collisions, overflows or under-utilization since the size of the hash table is hard to estimate precisely in some applications like database systems and file systems [36–38, 40]. Dynamic hashing [24], on the other hand, features in dynamically adjusting the size

of the hash table as needed by the *resizing* operation. In view of this, many delicately-designed dynamic hashing schemes are proposed for PM to achieve different optimizations, like PFHT [12], Path Hashing [49], Level Hashing [50], CCEH [36], Dash [33], and Clevel Hashing [10]. This work also focuses on the dynamic hashing schemes for PM.

Thanks to all of these efforts, the existing dynamic hashing schemes especially developed for PM have made remarkable progress on improving the overall *performance efficiency* in terms of mean throughput or mean latency. Nevertheless, surprisingly less attention has been given to the *performance predictability*, a particularly important metric in situations where the high-percentile performance would largely affect the quality of service (QoS) or the end-user experience [27, 28, 34]. In view of this, we conduct intensive experiments on a 24-core/48-thread CPU socket with six 128 GB Intel® Optane™ DCPMM to examine the in-depth performance of PM hashing schemes by utilizing different number of concurrent threads (ranging from 1 to 48). Our results (presented in §2.2) disclose that the representative hashing schemes for PM might 1) encounter the dilemma of simultaneously maintaining high *performance efficiency* and alleviating the resizing-incurred *performance unpredictability*, and 2) fall short of exhibiting good *performance scalability* under highly-concurrent queries due to their excessive writes in handling insert operations. Both seriously limit the practicality of existing PM hashing schemes to time-sensitive or latency-critical applications.

Aiming at developing a more practicable dynamic hashing scheme on PM, this paper present SEPH, a Scalable, Efficient, and Predictable Hashing for PM, to hit “three birds” with one stone. First of all, to break the dilemma between efficiency and predictability, SEPH introduces a new structure called *level segment (LS)* to build the hash table with a unique and delicate indexing mechanism (i.e., *level segment index* and *sliding bucket index*). Particularly, with the LS-based hash table structure, SEPH mitigates the inefficiency in probing items randomly, and embraces the incremental resizing (i.e., the split operation) to prevent other concurrent threads from being blocked. Second, SEPH further enables a *low-overhead split* operation to significantly suppress the resizing-incurred performance unpredictability: It not only reduces the number of KV items to be rehashed to one-third of an LS (i.e., *one-third splitting*) but even avoids the pointer dereference required to rehash a KV item for most of the time (i.e., *dereference-free rehashing*). Third, to achieve ever-higher efficiency and scalability while ensuring the correctness and crash consistency, SEPH devises a *semi lock-free mechanism* that requires a “nearly-minimal” amount of writes to handle an insertion. Our results show that SEPH performs better than the state-of-the-art hashing schemes from three perspectives. First, for efficiency, SEPH averagely achieves 2.12× higher throughput than EH-based hashing schemes, and even deliver 15.4× higher average throughput than level-based hashing schemes. Second, in terms of scalability, as the number of threads in-

creases from 24 to 48, the performance of SEPH still scales up noticeably whereas the other hashing schemes barely improve. Third, SEPH provides more reliable predictability by achieving 11.4×~19.3× lower tail latency. SEPH is implemented in C++ and is available<sup>1</sup> for public use.

The rest of this paper is organized as follows. §2 presents the background and motivation regarding this work. Next, §3 introduces the design details of SEPH. Finally, §4 demonstrates the evaluations results and §5 concludes this work.

## 2 Background and Motivation

### 2.1 Hashing Schemes for Persistent Memory

Due to various *structural designs*, different hashing schemes typically have their own way to perform the *resizing operation*, an essential but expensive operation entailing extra reads and writes to enlarge the hash table for accommodating more key-value (KV) items. The existing hashing schemes, especially developed for PM, can be generally categorized into two series: 1) **Level-based hashing**, a series that features a *multi-level structure* to enable *cost-efficient resizing*, and 2) **EH-based hashing**, a series that inherits the advantage of *incremental resizing* from Extendible Hash (EH) [13].

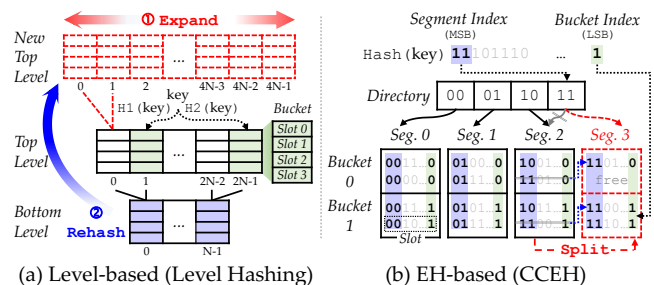


Figure 1: Two Series of Hashing Schemes for PM.

#### 2.1.1 Level-based Hashing

**Level Hashing.** Since memory writes in PM typically consume more time and energy than memory reads, the extra writes entailed by the resizing operation might bring a negative impact on PM in terms of both performance and endurance. Because of this reason, Level Hashing [50] introduces a new *sharing-based two-level structure* to enable a *cost-efficient resizing operation* for PM.

As illustrated in Figure 1(a), Level Hashing organizes KV items into two levels (i.e., *top level* and *bottom level*) of Bucketized Cuckoo Hashing (BCH) [14] with every bottom-level bucket shared by two consecutive top-level buckets, and thus the total number of buckets in the bottom level is just a half of that in the top level. Just like the design of BCH, Level Hashing employs a pair of hash functions (denoted as H1 and H2 in Figure 1(a)) so that any KV item can be associated with two buckets (aka *candidate buckets* [50]) in each level and can be placed in any *slot* of the candidate buckets. But in

<sup>1</sup><https://github.com/cuhk-mass/SEPH>

Level Hashing, only the top-level buckets are addressable by hash functions while a bottom-level bucket is mainly served as standby slots to keep conflicting KV items. That is, if a hash collision occurs in a top-level bucket and all slots in that bucket are used, the conflicting KV item can be stored in its corresponding standby bucket in the bottom level.

When all possible candidate buckets are full, Level Hashing cost-efficiently resizes the hash table as follows. As illustrated in Figure 1(b), the hash table is firstly “expanded” by allocating a *new top level* that is twice the size of the original top level and is with all the contents cleared with zeroes (denoted as ①); then, all the KV items in the original bottom level are “rehashed” into the newly allocated top level (denoted as ②); finally, the newly allocated top level and the original top level form a new sharing-based two-level structure. That is, with the cost-efficient resizing operation, Level Hashing doubles the total size of hash table but only rehashes and migrates KV items in one-third of buckets of the original hash table.

**Clevel Hashing.** Although Level Hashing enables a cost-efficient resizing operation, it must lock the entire hash table structure and inevitably blocks the normal hash operations (e.g., insert and search operations) from concurrent threads. To address this issue, Clevel Hashing [10], a *crash-consistent and lock-free concurrent hash table* is developed based on Level Hashing. Specifically, to avoid blocking the concurrent accesses during the resizing operation, in Clevel Hashing, the thread that triggers the resizing operation only expands the hash table for completing the insertion of KV item in the newly allocated top level, while the remaining work of rehashing is postponed and offloaded to dedicated background thread(s). As a result, in Clevel Hashing, the hash table may consist of more than two levels when it is under resizing.

### 2.1.2 EH-based Hashing

Another series of PM hashing schemes is evolved from Extendible Hashing (EH) [13], a widely-adopted hashing scheme that features the incremental resizing operation, called *split operation*, to avoid the full-table rehashing. Particularly, EH organizes KV items in *buckets* of a fixed number of *slots*, where a *directory* is maintained to index buckets based on the hashed value of a key (hereafter called the *hashed key* for simplicity). When a bucket overflows, EH performs the split operation to resize the hash table in the granularity of bucket rather than the entire hash table.

**Cacheline-Conscious Extendible Hashing (CCEH).** On the basis of EH, CCEH [36] is developed to make effective use of cachelines for better performance while guaranteeing failure-atomicity for dynamic resizing. Specifically, CCEH proposes to set the bucket size to the size of a cacheline (e.g., 64-byte) for minimizing the number of cacheline accesses for visiting a bucket. Besides, as shown in Figure 1(b), CCEH introduces an intermediate granularity named *segment*, which consists of a fixed number of buckets indexed by the same directory

entry, so that the directory can be greatly shrunk to have a higher probability of being in the CPU cache. CCEH also introduces a new way to associate KV items with segments and buckets: The most significant bits of the hashed key are used to locate a segment (denoted as *segment index*) while the least significant bits are used to index a bucket within a segment (denoted as *bucket index*). To further increase the load factor, CCEH adopts linear probing [15] so that a KV item can also be placed in the next few (e.g., four) buckets following the indexed one (by the bucket index).

When all candidate buckets (i.e., the indexed bucket and the following few that can be linearly probed) are all full for a newly-inserted KV item, CCEH resizes its hash table via the split operation (i.e., an incremental resizing operation introduced by EH) as follows: First, as illustrated in Figure 1(b), a new empty segment is dynamically allocated. Second, KV items in the collided segment are either stayed or rehashed into the newly allocated segment according to their segment and bucket indexes. Finally, after all KV items are rehashed, the directory is updated to ensure that the newly allocated segment will be indexed properly by the corresponding directory entry.

**Dynamic and Scalable Hashing (Dash).** Dash [33] further introduces several advancements to two classical hashing schemes (i.e., Extendible Hashing (EH) [13] and Linear Hashing (LH) [30]) and showcases its effectiveness on real PM product (i.e., Intel Optane DCPMM [3]).

Dash for EH (Dash-EH) inherits most of designs from CCEH [36] but aligns the bucket size with the XPLine size (i.e., 256-byte) of Intel® Optane DCPMM for better locality [33]. Moreover, Dash-EH divides every bucket into a record region (224-byte) and a metadata region (32-byte), where the former maintains pointers to KV items for supporting variable-length keys and values while the latter is dedicated to optimizing the probing and load factor. On the one hand, for every KV item, Dash-EH keeps the second least significant byte of the hashed key as a *fingerprnt* in the metadata region, so that the number of pointer dereferences, required by probing or checking the uniqueness of a KV item, can be thereby reduced; besides, Dash-EH adopts an optimistic concurrency control to avoid locking the entire segment when searching a KV item. On the other hand, Dash-EH combines a variety of techniques to increase the load factor, such as probing one more bucket, balancing the load factor of candidate buckets, allowing one movement among the indexed and linearly-probed buckets, and adding a few (e.g., two or four) *stash buckets* into each segment to accommodate conflicting KV items.

## 2.2 Motivation

Though the existing studies have made remarkable progress on advancing the hashing schemes for PM, this section will disclose that the existing two series of hashing schemes might 1) encounter the dilemma of achieving both high *performance efficiency* and high *performance predictability* simultaneously



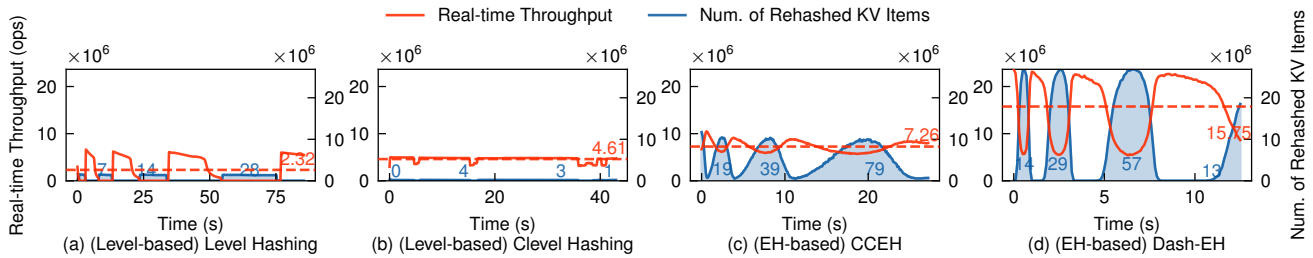


Figure 2: Real-time Throughput and Resizing Overhead under Mixed Workload (i.e., 50% Insertion and 50% Search).

(see §2.2.1) and 2) fall short of exhibiting good *performance scalability* under highly-concurrent queries (see §2.2.2). Both seriously limit the practicality of existing PM hashing schemes to time-sensitive or latency-critical applications.

### 2.2.1 Dilemma between Efficiency and Predictability

When examining the performance of PM hashing schemes, most of the existing studies mainly focus on the overall *performance efficiency* (i.e., the average performance) yet overlook the *performance predictability* (i.e., the high-percentile performance [27, 28, 34]), which is particularly crucial to time-sensitive or latency-critical applications in practice.

To investigate both performance efficiency and performance predictability of the existing two series of PM hashing schemes, we conduct intensive experiments on a 24-core/48-thread CPU socket with six 128 GB Intel® Optane™ DCPMM configured as the *App Direct* mode. More detailed experimental setups and implementation notes can be found in §4.1. Particularly, we preload each hashing scheme with 10 millions of KV items, and then measure the real-time performance of executing 200 millions of realistic mixed workloads with 48 concurrent threads, where the workloads consist of 50% search and 50% insert operations generated by YCSB [11] under the Zipf distribution with both key and value sizes set to 16B. Figure 2 shows the real-time throughput (in terms of operations per second) and the real-time resizing overhead (in terms of the number of rehashed KV items) of the four representative PM hashing schemes presented in §2.1. The results reveal that the existing PM hashing schemes might encounter the dilemma of achieving both high efficiency and high predictability simultaneously based on the following two key observations.

**Observation 1:** *Compared with Level-based hashing schemes, EH-based hashing schemes demonstrate the strength in performance efficiency yet entail heavier resizing-incurred overhead to degrade its performance predictability.*

It can be firstly observed from Figure 2 that EH-based hashing schemes demonstrate superior performance efficiency (i.e., at least 57.48% faster in terms of average throughput) than Level-based hashing schemes. The rationale behind this can be attributed to how these hash schemes probe the candidate buckets for a query. Specifically, EH-based hashing schemes probe the candidates buckets by sequential accesses, while Level-based hashing schemes entail one random access

for each of the candidate bucket (which is inherited from BCH [14]). Given that the latency of random read is about 1.8× longer than that of sequential read on PM (according to Table 1), it turns out that EH-based hashing schemes hold the advantage in performance efficiency.

Nevertheless, since EH-based hashing schemes naturally entail heavier resizing overhead (i.e., the number of rehashed KV items) than Level-based hashing schemes, their performance predictability can be affected more considerably. It can be clearly observed that the real-time throughput of EH-based hashing schemes gets degraded severely while KV items are being rehashed at that time; additionally, the more KV items are being rehashed, the lower throughput would suffer. Moreover, it is worth noting that, though Dash-EH utilizes a variety of techniques to postpone split operations for higher load factor, it may concentrate the occurrence of split operations as an adverse effect, leaving the performance predictability of Dash-EH unimproved or even degraded. As shown in Figures 2(c) and 2(d), when compared with CCEH, Dash-EH achieves 2.16X higher average throughput but suffers 5.78% lower worst throughput (i.e., the 100th percentile throughput).

**Observation 2:** *Compared with EH-based hashing schemes, Level-based hashing schemes entail lower resizing-incurred overhead yet still fail to deliver good performance predictability due to its low performance efficiency.*

As revealed by Figure 2, thanks to the cost-efficient resizing, Level-based hashing schemes greatly alleviate the total resizing overhead than EH-based hashing schemes. Cumulatively, Level-based hashing schemes incur at least 55.45% less number of rehashed KV items than EH-based hashing schemes after handling the same amount of insert operations. It is also worthy to note that, to avoid locking the entire hash table and blocking all the other concurrent requests during the resizing (as Level Hashing does), Clevel Hashing advocates a lock-free scheme and further postpones and offloads the rehashing of KV items to dedicated background thread(s), which explains why Clevel Hashing could incur even less number of rehashed KV items than Level Hashing in the evaluation.

However, unfortunately, the effective reduction in the resizing overhead is insufficient in helping Level-based hashing schemes with delivering good performance predictability. This is because Level-based hashing schemes suffer much worse performance efficiency, not only the average but also the worst ones, when compared with EH-based hashing schemes.

Specifically, even though the worst throughput (i.e., the 100th percentile throughput) of Clevel Hashing seems to drop less from its average throughput, it is still worse than that of CCEH and Dash-EH by 47.95% and 44.87% respectively.

### 2.2.2 Limited Scalability

Apart from the efficiency and predictability, the *performance scalability* is also an important indicator that reflects how efficient a hashing scheme is in processing concurrent requests. To this end, we repeat the experiments presented in Figure 2 with a different number of concurrent threads, ranging from 1 to 48, and show the measured average throughput of different hashing schemes in Figure 3.

**Observation 3:** *The existing PM hashing schemes fall short of exhibiting good performance scalability under highly-concurrent requests due to the excessive writes in handling insert operations.*

From Figure 3(a), it can be clearly observed that none of the evaluated hashing schemes could scale up the average throughput well from 24 concurrent threads. To find out the potential bottleneck to achieve good performance scalability, we further measure the total writes of PM media introduced by different hashing schemes (by reading the hardware counters of DCPMM [45]), since the write bandwidth is one of the major weaknesses of PM (according to Table 1). The results in Figure 3(b) identify that all the evaluated hashing schemes introduce more than twice amount of writes than expected (which was estimated by multiplying the number of insert operations by the XPLine size (i.e., 256-byte)), except the lock-free Clevel Hashing.

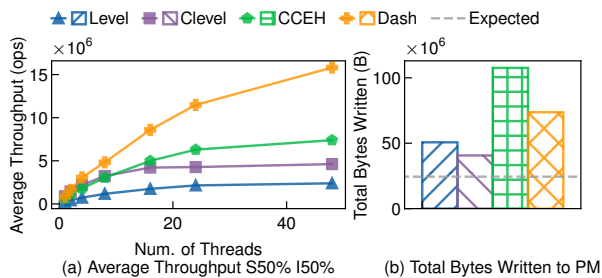


Figure 3: Scalability of Existing PM Hashing Schemes under Mixed Workload (i.e., 50% Insertion and 50% Search).

Based on our further investigation, such excessive amount of writes can be attributed to different root causes about how the PM hashing scheme handles an insertion. Specifically, for every insert operation, lock-based hashing schemes (such as Level Hashing, CCEH and Dash-EH) require one PM write to lock and insert the KV item, and another PM write for unlocking. As for a lock-free hashing scheme (like Clevel Hashing), it always requires one additional flush to persist its metadata, before every insertion, for the sake of crash consistency. In addition, it allows multiple threads to concurrently expand the hash table for the same level, but only one expansion would succeed eventually. This results in that the other failed expansions must waste PM writes to clear the memory space.

## 3 Design of SEPH

This section presents SEPH, a hashing on PM which can hit the *scalability*, *efficiency* and *predictability* with one stone. In this section, we first present a new structure called *level segment (LS)*, a key enabler to achieve both high efficiency and predictability, and elaborate on how SEPH builds a hash table based on LS (§3.1). Then, we show how LS can further enable a *low-overhead split* to greatly suppress the performance unpredictability caused by resizing (§3.2). Finally, we put forward a *semi lock-free concurrency control* that requires a *nearly-minimal* amount of writes to handle an insertion for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency (§3.3).

### 3.1 Level Segment based Hash Table

To resolve the dilemma between efficiency and predictability disclosed by §2.2.1, SEPH introduces a new structure called *level segment (LS)* to build the hash table by combining the respective strengths of the existing two series of PM hashing. Specifically, as we are going to see in this section, LS learns from EH-based hashing to achieve better efficiency in two ways: 1) LS limits the number of buckets that need to be randomly read for a query; and 2) LS enables the incremental resizing (i.e., the split operation) to avoid the full-table rehashing. Moreover, as we will elaborate in §3.2, LS further enables a low-overhead split operation, which is inspired by the two-level structure of Level-based hashing, to greatly harness the performance unpredictability caused by resizing.

#### 3.1.1 Structure

**Physical Segment.** To ease the dynamic memory allocation of PM space, SEPH manages the PM space as fixed-sized units called *physical segment (PS)*, which can be regarded as a “segment” in the EH-based hashing. To be more specific, a PS in SEPH also comprises a fixed number (e.g.,  $2^B$ ) of *buckets*, each bucket also consists of a fixed number of *slots*, and each slot can also accommodate one KV item. Besides, as suggested by Dash [33], SEPH also aligns the bucket size with the XPLine size (i.e., 256-byte) of Intel<sup>®</sup> Optane<sup>™</sup> DCPMM for achieving better locality.

**Level Segment.** To combine the respective strengths of the existing two series of PM hashing, SEPH further organizes PSs into a two-level structure called *level segment (LS)*, which is also the granularity for splitting. As depicted in Figure 4, given one PS at lower level (e.g., PS 0) and two PSs at higher level (e.g., PS 1 and PS 2), SEPH organizes the “left half” of lower-level PS and one higher-level PS into a LS (e.g., LS 0 which is denoted by blue-shaded region) and organizes the “right half” of the lower-level PS and another higher-level PS into the second LS (e.g., LS 1 which is denoted by green-shaded region). Moreover, within an LS, every two physically-consecutive higher-level buckets share a lower-level bucket, but SEPH gives higher priority to the lower-level

buckets for accommodating newly inserted KV items than the higher-level buckets for the sake of concurrency control (see §3.3). That is, only if all slots in a lower-level bucket of an LS are fully occupied, will a new KV item be inserted into the higher-level bucket of that LS. In view of this, when querying a KV item in an LS, SEPH also searches the lower-level candidate bucket before searching the higher-level candidate bucket for better search efficiency.

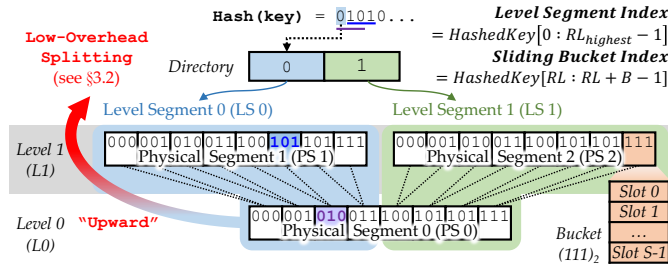


Figure 4: Level Segment based Hash Table of SEPH.

**Upward Splitting.** When the lower-level and higher-level candidate buckets are both full to a new KV item, SEPH *splits* that LS into two to enlarge the hash table in a copy-on-write (CoW) fashion (i.e., by rehashing existing KV items into newly allocated PSs). However, unlike the EH-based hashing splits horizontally, SEPH splits an LS in an “upward” and “low-overhead” fashion (see §3.2 for details). Consequently, as SEPH keeps splitting LSs to accommodate more KV items, the overall hash table will grow upwardly, since some LSs are evolved from more times of upward splitting and thereby reach higher levels than other LSs.

**Residing Level.** To precisely maintain *which level a PS currently resides*, SEPH associates every PS with an attribute called *residing level (RL)*. Taking the blue-shaded LS 0 depicted in Figure 4 as an example, the *RLs* of its higher-level PS and lower-level PS are 1 and 0, respectively.

### 3.1.2 Indexing

**Level Segment Index.** Like the EH-based hashing, SEPH also maintains a *directory* and utilizes the most significant bits to index an LS. Specifically, if the current highest residing level among all PSs is  $RL_{highest}$ , there must be  $2^{RL_{highest}}$  entries in the directory, and SEPH utilizes the first  $RL_{highest}$  most significant bits of the hashed key as the *level segment index*. Consider the example illustrated in Figure 4 where the current highest residing level  $RL_{highest}$  is 1. SEPH will associate the given KV item, whose hashed key starts with “01010”, with the LS indexed by the first directory entry (since the most significant bit in the hashed key is “0”).

**Sliding Bucket Index.** However, to facilitate the low-overhead split (introduced in §3.2), unlike the EH-based hashing that utilizes the least significant bits to index a bucket, SEPH introduces a unique *sliding bucket indexing* to index the candidate bucket in a PS for a KV item. Specifically, suppose  $RL$  denotes the *residing level* of a PS comprising  $2^B$  buckets,

SEPH uses the  $RL^{th}$  to  $(RL + B - 1)^{th}$  of the most significant bits in the hashed key (denoted as  $HashedKey[RL : RL + B - 1]$ ) to locate the candidate bucket according to the residing level  $RL$  of the PS.

Following the rule, SEPH can easily locate two candidate buckets (one in the lower-level PS and the other in the higher-level PS) in an LS for any KV item. As the example shown in Figure 4 where  $B$  is 3, given a hashed key starting with “01010”, the candidate bucket at lower level PS (e.g., PS 0 at Level 0) is the third one (e.g., Buckets  $(010)_2$ ) since  $HashedKey[0 : 2]$  is “010”, and the candidate bucket at higher level PS (i.e., PS 1 at Level 1) is the sixth one (i.e., Buckets  $(101)_2$ ) since  $HashedKey[1 : 3]$  is “101”.

## 3.2 Low-Overhead Split

To suppress the resizing-incurred performance unpredictability, SEPH proposes a *low-overhead split* operation, which not only reduces the number of KV items to be rehashed to one-third of an LS (i.e., *one-third splitting* in §3.2.1) but even avoids the pointer dereference required to rehash a KV item for most of the time (i.e., *dereference-free rehashing* in §3.2.2).

### 3.2.1 One-Third Splitting

With the novel Level Segment (LS) based hash table structure and the unique indexing mechanism (presented in §3.1), SEPH enables the *one-third splitting*, which only needs to rehash “one-third” of the KV items upon splitting an LS (i.e., the victim LS) into two new LSs as follows: ❶ Two new PSs are allocated at one level higher than the higher-level PS of the victim LS to address the hash collision; ❷ Only the KV items in the lower-level buckets (i.e., one-third) of the victim LS are rehashed into the two newly allocated PSs but the two newly allocated PSs and the KV items stayed in the original higher-level PS of the victim LS amazingly form two new LSs at one level higher, thanks to the unique level segment and sliding bucket indexes presented in §3.1; ❸ The corresponding directory entries are updated accordingly to point to the two newly formed LSs; ❹ The PM space occupied by the lower-level buckets of the victim LS is safely released.

Figure 5 depicts an example that walks through the whole process of the one-third splitting, where each PS is of 8 buckets (i.e.,  $B$  equals 3). Suppose we are going to insert a new KV item with the hashed key starting with “00011” into LS 0, but the two candidate buckets (i.e., Bucket  $(000)_2$  of PS 0 and Bucket  $(001)_2$  of PS 1) are both full. To address such hash collision, SEPH splits LS 0 by rehashing only its lower-level buckets into the two newly allocated PSs (i.e., PS 3 and PS 4) at Level 2. That is, with the unique level segment index and sliding bucket index, the KV items in Buckets  $(000)_2$  and  $(001)_2$  of PS 0 are rehashed into the newly allocated PS 3 while the KV items in Buckets  $(010)_2$  and  $(011)_2$  of PS 0 are rehashed into the newly allocated PS 4; however, there is no need to rehash any KV items in PS 1 since the two newly



allocated PSs (i.e., PS 3 and PS 4), along with the existing PS 1, amazingly form two new LSs (i.e., LS 2 and LS 3). At last, the directory entries are accordingly modified to index two newly formed LSs, and the “to-be-inserted KV item” can be eventually inserted into Bucket (011)<sub>2</sub> of PS 3 of the newly formed LS 2.

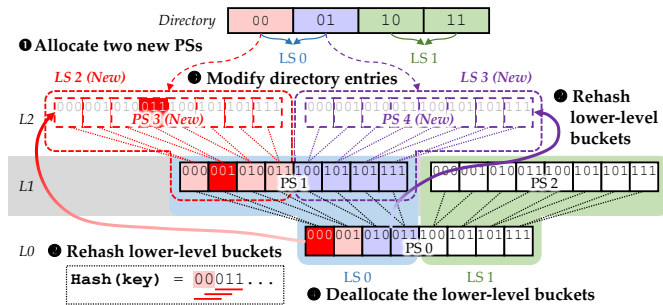


Figure 5: An Illustrative Example of One-Third Splitting.

### 3.2.2 Dereference-Free Rehashing

To support variable-length keys and values, like many representative PM hashing schemes (e.g., Clevel Hashing and Dash), SEPH keeps the pointers to KV items in slots of buckets. Consequently, to rehash a KV item (during the resize/split operation), typically, the pointer needs to be first *dereferenced* and a subsequent memory read is needed to get the content of a KV item, resulting in a considerable amount of random reads to degrade the performance on PM. In view of this, SEPH further enables the *dereference-free rehashing* that circumvents the pointer dereferences required to rehash KV items for minimizing the overhead of one-third splitting.

**Key Insight.** The main purpose of dereferencing a pointer during resizing is to locate the new candidate bucket a KV item based on the re-calculated hashed key. It means that if the new candidate bucket can be known by some means, a KV item can be directly moved into the new candidate bucket without dereferencing the pointer. Thanks to its unique sliding bucket indexing, SEPH can simply infer the new candidate bucket *if the two subsequent bits, following the current sliding bucket index of the hashed key, can be known*. This is because, during the one-third splitting, the KV items are always rehashed from the lower-level buckets of the victim LS into a newly allocated PS, which locates at two-level higher. To be more specific, for any KV items stored in the lower-level buckets of the victim LS residing at Level  $RL$ , its current sliding bucket index equals  $HashedKey[RL : RL + B - 1]$ ; since this KV item will be rehashed into a new PS located at two-level higher (i.e., Level  $RL + 2$ ), its new sliding bucket index will become  $HashedKey[RL + 2 : RL + B + 1]$ . In other words, SEPH can infer the new candidate bucket for this KV item by only requiring two extra bits, i.e., the  $(RL + B)^{th}$  and  $(RL + B + 1)^{th}$  bits in its hashed key.

**Bucket Index Foreseer.** Based on this key insight, SEPH proposes to maintain a small chunk of the hashed key, called

*bucket index foreseer* (or *foreseer* for simplicity), which contains the required “two bits” for dereference-free rehashing, along with the pointer to that KV item in the slot. In our implementation, the size of the foreseer is set to 16 bits since the modern 64-bit operating systems typically use 48 or fewer bits of pointers. As shown in Figure 6, when inserting a new KV item into a PS of  $2^5$  buckets residing at Level 0, SEPH keeps not only the pointer to this KV item but also the first two bytes of the hashed key (i.e., “00101010 10101101”) as the foreseer in the 64-bit slot. Later, when this KV item needs to be rehashed into a newly allocated PS at Level 2, since the 6<sup>th</sup> and 7<sup>th</sup> bits of the hashed key (i.e., “01”) are maintained in the foreseer, SEPH can directly move this KV item into Bucket (10101)<sub>2</sub> in the new PS without dereferencing the pointer (denoted by ❶ in Figure 6).

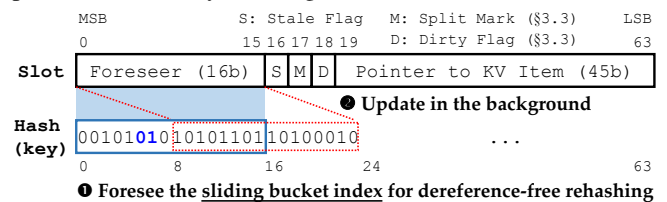


Figure 6: An Running Example of Bucket Index Foreseer.

Nevertheless, with the growth of the hash table, the foreseer might contain fewer and fewer “unused” bits and eventually “fail to foresee” the new sliding bucket index during the subsequent split operations. Thus, SEPH proposes to keep the number of unused bits in the foreseer more than half (i.e., 8 bits) at most times by updating the foreseer in the background (denoted by ❷ in Figure 6). To this end, SEPH maintains another bit called *stale flag* in the slot to indicate the staleness of the foreseer. If the unused bits of the foreseer will be less than half of its size after a dereferente-free rehashing, SEPH sets the stale flag to 1 and submits a job to a dedicated thread, which can update the foreseer and reset the stale flag, via an atomic operation without consistency issue, in the background. Notably, if all the unused bits in a foreseer have really been used up without being timely updated, SEPH alternatively updates the foreseer right away in the foreground before rehashing the KV item.

It is also worth mentioning that the foreseer can also be utilized as the “tag” in Clevel Hashing [10] or “fingerprint” in Dash [33] to avoid unnecessary dereferences of pointers during bucket probing. This is because the foreseer will get updated timely to contain bits, which are neither LS nor sliding bucket indexes, so that it is particularly effective in telling whether a KV item exists in a bucket.

### 3.3 Semi Lock-Free Concurrency Control

As discussed in §2.2.2, the existing PM hashing schemes introduce an excessive amount of writes to handle an insertion. This not only brings considerable degradation to efficiency, but even largely limits the scalability especially under highly-concurrent and insert-intensive scenarios. To



achieve ever-higher efficiency and scalability, SEPH proposes a *semi lock-free concurrency control* that requires a nearly-minimal amount of writes to handle an insertion. §3.3.1 and §3.3.2 shall first introduce the design concept and the correctness challenges of the semi lock-free concurrency control, respectively. Then, §3.3.3 elaborates on how to conduct different hash operations while guaranteeing the correctness and crash consistency in depth.

### 3.3.1 Design Concept

Thanks to its atomicity, the compare-and-swap (CAS) primitive [4] has been widely adopted by many existing lock-free data structure and algorithm designs to allow concurrent operations without explicitly managing locks [22, 32, 39, 41]. The CAS primitive “compares” the stored content of a word in memory with a given value and, only if they match, “swaps” the content of that word with the given value; meanwhile, a Boolean value is returned to indicate whether the swap takes place or not. Notably, the execution of the CAS primitive is guaranteed to be atomic in the sense that the content of the memory word is either completely swapped or stays unchanged in an “all-or-nothing” fashion. Since SEPH sets the slot size to the word size (e.g., 8 bytes) (see §3.2.2), we can also leverage the CAS primitive to realize lock-free operations for the avoidance of excessive amounts of writes to PM due to the lock manipulation. However, in view of the fact that the split operation occurs relatively infrequent but could complicate the correctness guarantee of other frequent hash operations (i.e., insert/update/delete/search operations) [10], we propose to prudently apply the lock only to the split operation. That is, SEPH puts forward a *semi lock-free concurrency control* in which only the (infrequent) split operation needs to acquire the lock for splitting an LS, while other (frequent) hash operations (i.e., insert/update/delete/search operations) are all lock-free even when the involved LSs are under splitting.

### 3.3.2 Correctness Challenges

Guaranteeing the correctness of concurrent executions is one of the most critical challenges when designing lock-free data structures or algorithms. However, even though the CAS primitive can guarantee the atomicity of manipulating a slot, according to [10], performing hash operations in a lock-free manner may still lead to two correctness problems, i.e., *duplicate items* and *loss of items*.

**1) Duplicate Items.** The correctness problem of *duplicate items* is that concurrent lock-free insertions may place multiple KV items with the same key into different slots of the hash table. This may violate the correctness of subsequent update/delete/search operations, since the update or delete operation may only take place in one slot while the search operation may access other duplicate slots that are unmodified.

**2) Loss of Items.** The correctness problem of *loss of items* is that the modifications to the hash table (made by insert/update/delete operations) may be lost when the hash

table is under resizing concurrently. This is possible since the concurrent modifications may be left behind (i.e., not rehashed) by the resizing, making those non-rehashed modifications “invisible” to the subsequent operations incorrectly.

### 3.3.3 Operation Details

**Lock-Free Insert.** In SEPH, the insertion operation is designed to be *lock-free* for the avoidance of the concurrency control overhead in manipulating locks. In order to address the correctness problem of *duplicate items* (see §3.3.2) caused by concurrent insertions, SEPH regulates the order of empty slot allocation in the two candidate buckets to accommodate newly inserted KV items based on the following two rules of thumb: 1) The slots in the lower-level candidate bucket must be first used up before using the slots in the high-level candidate bucket. 2) In a candidate bucket, the slots must always be allocated from the first one to the last one, where no empty slots can exist before any allocated slots and no deleted slots can be re-allocated. In summary, together with the two rules and the atomicity of CAS primitive, SEPH guarantees that concurrent insertions to the same LS will always compete for not only the same candidate bucket (rule 1) but also for the same empty slot in that bucket (rule 2) so that the correctness problem of duplicate items can be nicely avoided.

Algorithm 1 elaborates the lock-free insert operation in detail. First, it looks up the directory to find out the corresponding LS for the to-be-inserted KV item (Line 2) and performs a uniqueness check to ensure that in the two candidate buckets of that LS, there are no existing KV items holding the same key as the to-be-inserted KV item (Lines 3–9). Specifically, the uniqueness check employs the atomic load instruction [2, 10, 17] to atomically fetch every allocated slot one after the other for examination (Line 5) and rejects an insertion request if its key matches the key of any allocated slots in the LS (Lines 8–9). Please note that the atomic instructions will not always lead to direct accesses to PM, since these requests can also be served in the cache [2].

Then, it starts to compete for the empty slot in the two candidate buckets based on our rules for the empty slot allocation (Lines 10–21). That is, the lower-level bucket is used before using the high-level bucket (rule 1) and the slots in a bucket are allocated from the first one to the last one (rule 2), so that all the concurrent insertions to the same LS will be regulated to always compete for the same empty slot. Especially, the CAS primitive is utilized to atomically check a slot is empty (by *comparing* the content of slot with an “empty” value) and fill in the slot (by *swapping* the content of slot with the pointer to the to-be-inserted KV item) (Line 12). Thanks to the atomicity of CAS primitive, even if there are concurrent insertions competing for the same empty slot, only one thread can successfully fill in it; then, the only “CAS-succeeded” thread utilizes the `c1wb` [2] and `mfence` instructions [2] to persist the filled slot into PM [7, 10, 33, 36] (Lines 13–15). Meanwhile, all the other “CAS-failed” concurrent threads must check every slot they failed to fill in, since those slot(s)

---

**Algorithm 1: LOCK-FREE INSERT**

---

**Input:** *kv*: the pointer to the to-be-inserted KV item  
**Output:** the result of insertion (*SUCCESS* or *FAIL*)

```
1  RETRY;  
2  Look up the directory to find out the LS for kv;  
   // uniqueness check  
3  for bucket in [LS -> PSlower, LS -> PShigher] do  
4    foreach slot in bucket do  
5      slot' ← ATOMIC LOAD(slot);  
6      if slot' has a set split mark then  
7        goto RETRY;  
8      if kv and slot' have the same key then  
9        return FAIL;  
   // CAS insert  
10 for bucket in [LS -> PSlower, LS -> PShigher] do  
11  foreach slot in bucket do  
12    resultCAS ← CAS(&slot, empty, kv);  
13    if resultCAS is successful then  
14      Persist slot into PM;  
15      return SUCCESS;  
16    else  
17      slot' ← ATOMIC LOAD(slot);  
18      if slot' has a set split mark then  
19        goto RETRY;  
20      if kv and slot' have the same key then  
21        return FAIL;  
   // split (both cand. buckets are full)  
22 Perform LOCK-BASED ONE-THIRD SPLIT() to split LS;  
23 goto RETRY;
```

---

may be inserted with KV items with the same key (Lines 16–21). If so, the insertion must be rejected to avoid duplicate items (Lines 20–21); otherwise, the CAS-failed thread(s) will continue to compete for the next empty slot iteratively.

Finally, if all the slots in the two candidate buckets are used up, the thread needs to trigger the one-third split operation (see Algorithm 2) to split the *LS* (Line 22), followed by retrying the insertion in a lock-free way (Line 23).

**Lock-Based One-Third Split.** In SEPH, the one-third split operation is *lock-based*. That is, an *LS* could only be split by one of the concurrent threads successfully. Even so, SEPH may still be threatened by the correctness problem of *loss of items* introduced in §3.3.2. Particularly, as the one-third split operation is rehashing the KV items from the lower-level PS of an *LS* into newly allocated PSs, some other concurrent lock-free /deletion operations may be making changes to that lower-level PS (since their lower-level candidate buckets are still not full), leaving some of these modifications not rehashed correctly. To resolve this correctness problem, SEPH devises a lightweight mechanism that allows a split operation to timely notify other concurrent lock-free operations of the rehashing status of every slot. Specifically, SEPH reserves a one-bit “split mark” in every slot, and the split mark will only be set atomically once the split operation has started to process it. It ensures that other concurrent lock-free operations can avoid

---

**Algorithm 2: LOCK-BASED ONE-THIRD SPLIT**

---

**Input:** *LS*: the *LS* that needs to be split

```
1  if TRY ACQUIRE LOCK(LS) == SUCCESS then  
2    Allocate two new PSs for one-third splitting;  
3    foreach bucket in LS -> PSlower do  
4      foreach slot in bucket do  
5        do // CAS Loop  
6          slot' ← ATOMIC LOAD(slot);  
7          slot'm ← slot' | split mark;  
8          resultCAS ← CAS(&slot, slot', slot'm);  
9          while resultCAS is not successful;  
10         if slot is not an empty or deleted slot then  
11           Perform DEREFERENCE-FREE REHASH() to  
             rehash slot into the two new PSs;  
12         Persist the two new PSs into PM;  
13         Form two new LSs and update the directory;
```

---

making changes to slots with a set split mark for the avoidance of loss of items, since the “compare” of CAS primitive will fail due to the set split mark bit.

As shown in Algorithm 2, the one-third split operation in SEPH needs to first acquire the lock for splitting any *LS* (Line 1), and only the thread successfully acquired the lock can rehash KV items from the lower-level PS of the to-be-split *LS* to the two newly allocated PSs (Lines 2–13). Especially, for every slot (including empty slots), the split operation shall first exploit the CAS loop [17] to ensure the split mark can be successfully set even in the presence of the concurrent operations (Lines 5–9). Then, the *dereference-free rehashing* (see §3.2.2) is employed to rehash the KV item if it exists (Lines 10–11). Finally, only after all the slots have been set with the split mark and all the KV items have been successfully rehashed and persisted into PM (Lines 12), should the directory entry be updated to index the two newly formed *LS*s (Line 13). It is the key step to ensure that no other concurrent operations can access the slots in the two newly allocated PSs when the splitting is still taking place. Notably, there is no need to release the split lock after splitting, because the split *LS* would become stale and any other concurrent threads should not split it again. Besides, we adapt the epoch-based reclamation [16] to recycle the stale PS only after no other concurrent lock-free readers are using it [33].

With the split marks, the problem of loss of items can be avoided as follows. Particularly, if the insertion ends with a CAS success, it implies that the concurrent split operation has not yet set the split mark for that slot and will rehash the inserted KV item later. Otherwise, if the insertion ends with a CAS failure because of a set split mark, it means that a concurrent split operation has already started to process this slot by first setting the split mark with the atomic CAS primitive (i.e., Line 8 in Algorithm 2). Thus, SEPH shall retry the entire insert operation to avoid leaving an insertion of KV item behind the concurrent split operation (i.e., Lines 18–19 in Algorithm 1). It is also worthy to note that during the uniqueness check, if a slot with a set split mark is found,

SEPH shall also retry the entire insert operation to avoid accessing stale KV items (i.e., Lines 6–7 in Algorithm 1); in addition, both two candidate buckets shall be examined, since the higher-level candidate bucket may have become the lower-level candidate bucket due to concurrent splits.

Notably, the problem of duplicate items can also be avoided even if an insertion ends with a CAS success but has time overlap with a split operation to the same LS. Let's first discuss the situation that the insertion can successfully fill in a slot in the "lower-level" candidate bucket of the LS. In this scenario, the split operation must be not over yet (since the split mark of this slot has not been set) and all the other concurrent insertions with the same key must fail to compete for the same slot in the same lower-level candidate bucket (thanks to the atomicity of CAS primitive). Next, let's discuss the other situation where the insertion can successfully fill in a slot in the "higher-level" candidate bucket (denoted as  $b$ ) of the LS (i.e., the corresponding lower-level candidate bucket is already full). In this scenario, the split operation may still be in the process or may have been completed by the time that slot is filled in. Specifically, if the split operation is not over yet, all the other concurrent insertions with the same key must compete in the same higher-level candidate bucket  $b$  but fail eventually. If the split operation is complete already, all the other concurrent same-key insertions launched before the split completion must compete in the same high-level candidate bucket  $b$  but fail eventually; meanwhile, since the higher-level candidate bucket  $b$  has become the lower-level candidate bucket in the new LS after the split completion (see §3.2.1), all the other concurrent same-key insertions with the same key launched after the split completion shall first compete in the same candidate bucket  $b$  (based on the rule 1 of empty slot allocation) but fail eventually.

**Lock-Free Update/Delete.** With the help of CAS primitive, in SEPH, the update and delete operations are also designed to be *lock-free*. To locate the KV item for update or deletion in the candidate buckets, the atomic load instruction is utilized (similar to how the uniqueness check is performed in the lock-free insertion). Then, if the slot with the desired key can be found, SEPH takes advantage of the atomicity of the CAS primitive to update the slot so that only one current thread can successfully modify it at a time. However, if the CAS primitive fails due to a set split mark in the slot, SEPH shall retry the entire update/delete operation to avoid leaving modifications to slots that have been processed by a concurrent split operation for the avoidance of loss of items.

On the other hand, based on the rules of thumb for empty slot allocation, no empty slots can exist before any allocated slots and no deleted slots can be re-allocated. Thus, in SEPH, the delete operation is realized in a way very similar to the update operation. The only difference is that the deletion replaces the desired slot with a "tombstone" instead of the updated KV item. In our implementation, we consider a slot that has all 1s for its pointer to KV item as a tombstone slot.

By doing so, a tombstone slot can be easily identified and more importantly, we can still exploit the split mark and the CAS primitive to avoid losing deletions in slots that have been processed by a concurrent split operation (as how we avoid losing updates).

**Lock-Free Search.** Since SEPH takes advantage of the atomicity of CAS primitive to modify a slot and utilizes the atomic load instruction to atomically read a slot, the search operation can be easily realized as *lock-free*. However, to avoid reading stale KV items, SEPH shall retry the search operation if any slot with split mark is accessed (as what we do in the uniqueness check of insertion).

### 3.3.4 Persistence for CAS

The compare-and-swap (CAS) atomic instruction [4] achieves the synchronization in multithreading; however, since the processor cache is typically volatile, a thread might access data that have not been persisted yet, resulting in data inconsistency in the presence of crashes. In our current implementation of SEPH, we utilize the *persistent single-word compare-and-swap (PSwCAS)* [42] primitive that can address this problem by adding a dirty bit on each 8-byte word operated by the CAS instruction. Specifically, the PSwCAS primitive requires that 1) the CAS instruction always stores a word of data with the dirty bit set; and 2) a thread must first persist the required word into PM if the word is set with the dirty bit, followed by clearing the dirty bit to mark the word as persistent.

Notably, the extended asynchronous DRAM refresh (eADR), a new feature supported by Intel<sup>®</sup> Optane<sup>™</sup> DCPMM 200 series and 3rd Xeon<sup>®</sup> Scalable processors, ensures that CPU caches are also included in the power fail protected domain [44]. That is, with the eADR technique, the CAS primitive can be used directly with the data consistency guarantee even in the presence of crashes [1]. Thus, we envision that SEPH shall be greatly benefited by the eADR feature to deliver even superior performance.

### 3.3.5 Crash Consistency

No inconsistency will occur in SEPH against crashes for the following reasons: 1) The insertion/update/deletion can be done atomically and their crash consistency can be guaranteed by the PSwCAS. 2) The split operation is protected by the lock and is conducted in a copy-on-write (CoW) manner, so the split operation is an all-or-nothing process; moreover, an unfinished split operation (which is broke off by the occurrence of crash) can also be identified (by examining the split locks of LS) and correctly redone. 3) The crash consistency for the directory can be secured by the directory recovery algorithm proposed in CCEH [36].

## 4 Performance Evaluation

### 4.1 Experimental setup

**Environment.** All experiments are conducted on a 24-core/48-thread Intel Xeon Platinum 8260 2.40 GHz CPU socket with

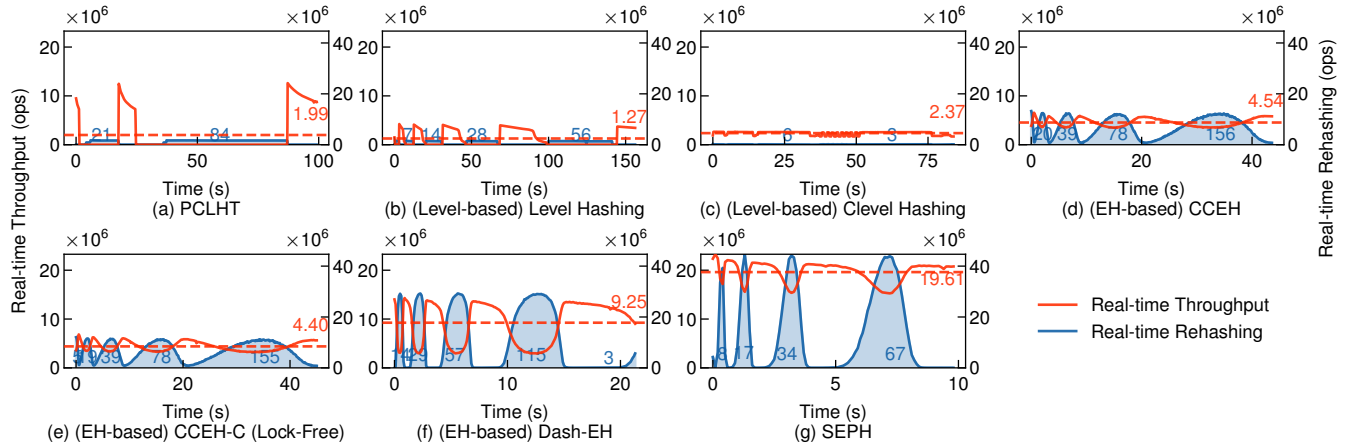


Figure 7: Real-time Throughput and Resizing Overhead of Insertion.

six 32 GB DRAM and six 128 GB Intel<sup>®</sup> Optane<sup>™</sup> DCPMM 100 series configured as the *App Direct* mode. The operating system is 64-bit Ubuntu Server 22.04 with Linux kernel version 5.15, and Persistent Memory Development Kit (PMDK) version is 1.11. All the codes are implemented in C++ and compiled using GCC 11.2 with all optimizations enabled.

**Evaluated Hashing Schemes.** We evaluate the following hashing schemes especially developed for PM, and adopt parameter settings suggested by their original papers for achieving the best performance on Intel<sup>®</sup> Optane<sup>™</sup> DCPMM 100 series.

- **PCLHT:** PCLHT is a search-optimized hashing scheme adopting a linked list based cache-efficient hash table converted by RECIPE [26].
- **Level1:** Level Hashing [50] is the origin of Level-based hashing schemes (see §2.1.1). It uses 128-byte buckets (i.e., two cachelines).
- **Clevel1:** Clevel Hashing [10] is an extension of Level Hashing with lock-free concurrency control (see §2.1.1). It uses 64-byte buckets (i.e., one cacheline) and employs one dedicated background thread to perform the resizing.
- **CCEH/CCEH-C:** CCEH [36] is developed based on Extendible Hashing (EH) [13] with effective use of cachelines for better performance (see §2.1.2). It uses 16 KB segments and 64-byte buckets (i.e., one cacheline) with a probing distance of 4. CCEH-C is a variant of CCEH that conducts the split operation in a CoW way to support lock-free search [36].
- **Dash:** Dash-EH [33] is an enhanced version of CCEH [36] with several technique advancements (see §2.1.2). It uses 16 KB segments and 256-byte buckets (i.e., an XPLine) with two additional stash buckets.
- **SEPH:** This is our proposed hashing scheme. To compress two PS pointers into one 8-byte word, we implement a segment allocator that supports atomic aligned segment allocation and crash consistency for SEPH. Besides, we set the size of a PS to 16 KB (which is also the segment size of CCEH and Dash used in the evaluation), and thus, the total size of an LS is 24 KB. Moreover, SEPH employs one dedicated background thread to update the bucket index foreseer.

For the sake of fairness, since the more recent PM hashing schemes (e.g., Clevel Hashing, Dash, and our proposal) support variable-length keys and values, all the evaluated hashing schemes are unified to only keep the pointers to KV items in slots. Besides, the length of a persistent pointer in PMDK is 16 bytes (i.e., 8 bytes for the base address of a PM pool and 8 bytes for the offset in pool), which cannot be operated by the CAS atomic instruction. To resolve this issue, Clevel Hashing [10] proposes to only maintain the offset in the PM pool as an 8-byte persistent pointer, since the base address of a PM pool will be fixed once the pool is mapped. We also apply this offset-only pointer to all the evaluated hashing schemes.

**Benchmark.** For the micro-benchmarks used in §4.2, we first warm up the hash table with 10 millions of KV items, followed by executing a total number of 200 millions of operations unless otherwise stated. Particularly, workloads composed of a single type of operations are evaluated in §4.2.1~§4.2.3 and workloads mixed with multiple types of operations are used in §4.2.4, where all these workloads are generated by YCSB [11] in Zipf distribution with 0.99 skewness. As for the macro-benchmarks presented in §4.3, we use the real-world workloads from YCSB [11]. Particularly, in the load phase, we populate 64 millions of KV items, following Clevel Hashing [10]; then, the standard YCSB workloads A, B, C, D, and F are conducted with 48 threads. Notably, the standard YCSB workload E is not evaluated since none of the hashing schemes optimizes the range query performance. Besides, the lengths of key and value are both set to 16 bytes since it is widely used [6], and the KV items are pre-generated before the testing as [33].

## 4.2 Micro Benchmark

### 4.2.1 Performance Efficiency and Predictability

To analyze the advantages of SEPH, we first focus on the insertion performance. Figure 7 plots the real-time insertion throughput of different hash tables under 48 threads. It can be clearly observed that SEPH outperforms all the other schemes



in terms of performance efficiency. Specifically, SEPH outperforms Dash, CCEH, and CCEH-C by 2.12 $\times$ , 4.31 $\times$ , and 4.46 $\times$  for average insertion throughput respectively, and achieves at least 8.27 $\times$  higher average throughput compared with Level-based hashing schemes and PCLHT.

As for the performance predictability, SEPH demonstrates the most superior worst-case real-time throughput (i.e., the minimal real-time throughput) than all the other evaluated hashing schemes, as revealed by Figure 7. Specifically, the minimal real-time throughput of SEPH is higher than that of Clevel, CCEH, CCEH-C, and Dash by 9.34 $\times$ , 4.40 $\times$ , 4.74 $\times$  and 5.23 $\times$  respectively, while PCLHT and Level Hashing even suffer zero real-time throughput because their full-table resizing are conducted in a blocking manner. Moreover, it is also worth noting that the minimal real-time throughput of SEPH is even higher than the maximal real-time throughput of all the other evaluated hashing schemes by from 1.06 $\times$  to 5.76 $\times$ . This reveals that SEPH achieves remarkable performance predictability by delivering an excellent worst-case real-time throughput under the insertion-intensive workload.

The tail latency of different percentiles is another perspective to show the performance by reflecting the response time. A design with good performance predictability requires a low bound of the latency on high percentiles (i.e., tail latency). Figure 8 shows the evaluation of the insertion latency at different percentiles. In general, SEPH significantly cuts down the 100th-percentile insertion latency compared with PCLHT/Level-based hashing schemes (by 3 ~ 4 orders of magnitude) and is superior to all the EH-based hashing schemes on every percentile of insertion latency. Especially, it can be noticed that in contrast to EH-based hashing schemes (i.e., CCEH, CCEH-C and Dash) that have a sharp raising of insertion latency at the 99.9th percentile, the insertion latency of SEPH rises at the 99.99th percentile. The rationale behind this is that SEPH triggers a less number of split operations than EH-based hashing schemes, since the size of LS in SEPH (i.e., 24 KB) is larger than the segment size (i.e., 16 KB) of EH-based hashing schemes. Despite this, SEPH still achieves 9.75 $\times$  ~ 11.36 $\times$  lower latency at both 99.99th and 99.999th percentiles than EH-based hashing schemes; Also, the 100th-percentile latency of SEPH is lower than that of EH-based hashing schemes by from 3.62 $\times$  to 5.86 $\times$  because SEPH offloads the directory doubling to the background thread.

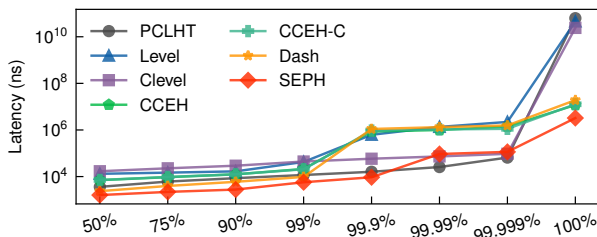


Figure 8: Latency at Different Percentiles.

Figures 9(a) and 9(b) further disclose the key reasons behind the improvements in performance efficiency and predictability

achieved by SEPH. On the one hand, thanks to the one-third splitting and the dereference-free rehasing, SEPH introduces 8.11 ~ 43.78 $\times$  less time for resizing than all the other evaluated hashing schemes as shown in Figure 9(a). On the other hand, Figure 9(b) validates the efficacy of the semi-lock-free concurrency control in minimizing the PM writes. Specifically, SEPH significantly reduces the PM writes by 2 ~ 3.33 $\times$  and nearly approaches the expected, optimal amount of PM writes.

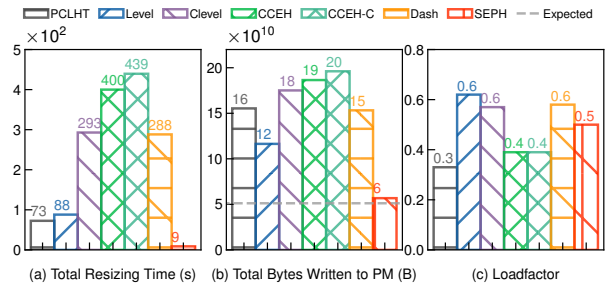


Figure 9: Resizing Time, Total Bytes Written, and Load Factor.

**Load Factor.** Figure 9(c) further depicts the result of average load factors. Considering that Dash leverages non-trivial memory PM (i.e., 12.5%) to store the metadata to improve the performance, the calculation of the load factor of Dash excludes this amount of memory. The average load factor of Level Hashing and Dash are both up to 59% because they adopt multiple optimizations such as one movement, balancing load, stash buckets, etc. The average load factor of SEPH is 50%, which is because SEPH does not adopt the optimization that improves the load factor at the expense of introducing more operational overhead. On the other hand, our design shows higher load factor compared to CCEH and CCEH-C (both 39%) since SEPH can accommodate more KV items in a bucket (i.e., 32 for SEPH).

#### 4.2.2 Performance Breakdown

To investigate how the key techniques of SEPH introduced in §3.2 and §3.3 contribute to the overall performance improvement, Figure 10 shows the experiment results of inserting 200 millions of KV items using 48 threads on different variants of SEPH (see Table 2 for their detailed configurations).

Table 2: Configuration of Different SEPH Variants.

SEPH Variants	Semi Lock-Free	One-Third Splitting	Dereference-Free Rehasing
SEPH-Base	×	×	×
SEPH-S	✓	×	×
SEPH-SO	✓	✓	×
SEPH-SOD	✓	✓	✓

First of all, SEPH-Base is considered as a baseline design of SEPH which only keeps the level segment based hash table of SEPH (presented in §3.1) but does not equip with any low-overhead split techniques (proposed in §3.2) and even adopts the lock-based scheme of Dash [33]. It can be clearly observed from Figure 10(a) that compared with SEPH-Base, SEPH-S (which adopts only the proposed semi lock-free concurrency control) significantly lifts up the average

and highest throughputs by 56.72% and 70.20% respectively. The rationale behind this is that the proposed semi lock-free concurrency control effectively avoids the PM writes required to manipulate locks (as validated by Figure 10(b)), resulting in better performance efficiency and scalability. Secondly, compared with SEPH-S, SEPH-SO further demonstrates the efficacy of the proposed one-third splitting in raising the worst-case throughput by 71.46% and reducing the total resizing time by 51.04% (as shown in Figure 10(c)). Finally, compared with SEPH-SO, SEPH-SOD (i.e., the complete design of SEPH) ultimately shows how the proposed dereference-free rehashing can amazingly minimize the total resizing overhead by 92.50%, resulting in a further improvement in the worst-case throughput (i.e., performance predictability) by 55.85%.

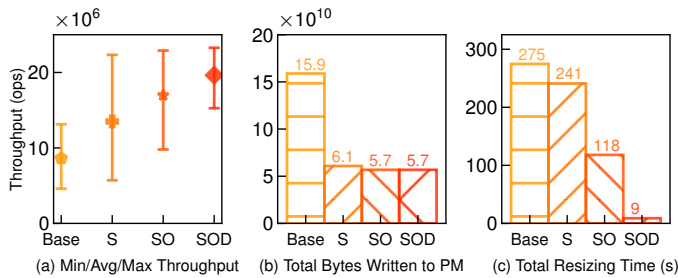


Figure 10: Breakdown of Different SEPH Variants.

### 4.2.3 Performance Scalability

**Insertion.** Figure 11(a) shows the insertion throughput under different number of threads. SEPH speeds up the insertion performance by  $2.2\times$  under 48 threads and by  $2\times$  under the other thread numbers compared with Dash. The main reason is that the write bandwidth of PM is considered a common bottleneck, and SEPH completes the insertions with less consumption of write bandwidth. By contrast, PCLHT and Level Hashing show poor scalability owing to the blocking resizing, while the low scalability of Clevel Hashing is due to the high consumption of read/write bandwidth for the full lock-free design.

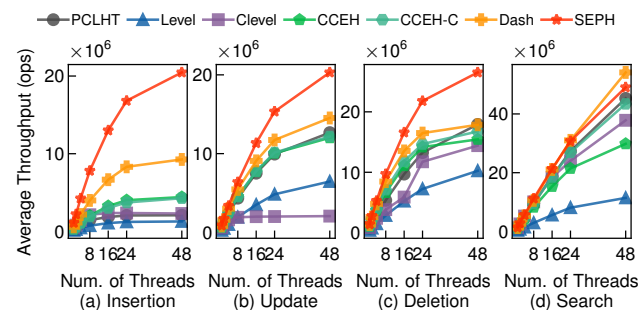


Figure 11: Scalability.

**Update and Deletion.** Figures 11(b) and 11(c) present the update and deletion performance of various hashing schemes under different numbers of thread. Although SEPH shows similar performance on update operations with Dash when the number of the threads is less than 16, SEPH outperforms Dash

by 30% at 24 threads and 39% at 48 threads, demonstrating higher performance scalability. This is because the lock-free design of SEPH provides a more scalable update performance by reducing a write for the lock to avoid hitting the limit of the write bandwidth at lower concurrent scenarios. To test the scalability of delete operations, we run 10 millions of delete operations to delete all the pre-loaded 10 millions of KV items. As shown in Figure 11(c), SEPH also surpasses all the other designs at 48 threads in delivering high performance scalability of deletion, thanks to the reduction in PM write achieved by the semi lock-free concurrency control.

**Search.** As shown in Figure 11(d), Dash, SEPH and PCLHT show good scalability on search performance, thanks to the low-overhead search operations of these designs and the high bandwidth of PM read (compared with write bandwidth). The search throughput of SEPH is 9.1% lower than that of Dash with 48 threads because SEPH needs to access two candidate buckets by two random reads, yet Dash accesses two candidate buckets by two sequential reads.

### 4.2.4 Mixed Workload

In order to evaluate the performance behavior of the different hashing schemes under the realistic mixed workload, we conduct the experiments with the mixed requests of different search/insertion ratio generated by YCSB in the zipfian distribution (0.99 skewness).

Figure 12 shows the real-time throughput of different hashing schemes with different mixed workloads under 48 threads. SEPH performs the mostly-highest and the least-fluctuated performance in these workloads, which demonstrates SEPH can achieve good performance efficiency and good performance predictability at the same time under the evaluated workloads mixed with different percentages of search operations (denoted as “S”) and insert operations (denoted as “I”).

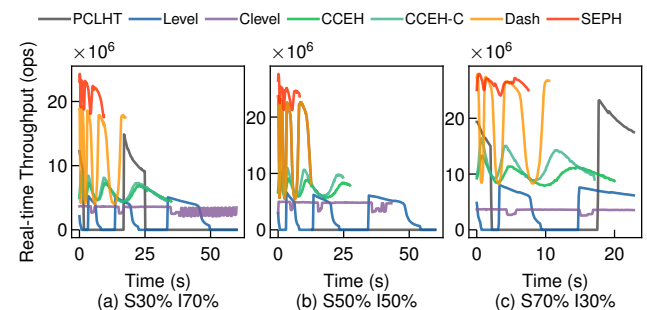


Figure 12: Real-Time Performance under YCSB Workloads with Different Search/Insert Ratio (%).

Figure 13 shows the results of the scalability of the hash tables under different mixed workloads. It can be observed that SEPH delivers better performance scalability than any other evaluated hashing schemes, even under the workload mixed with a high percentage of insertions (i.e., 70% of insert operations and 30% of search operations shown in Figure 13(a)). This is because the proposed semi lock-free

concurrency control entails a nearly-minimal amount of PM writes to handle an insertion operation.

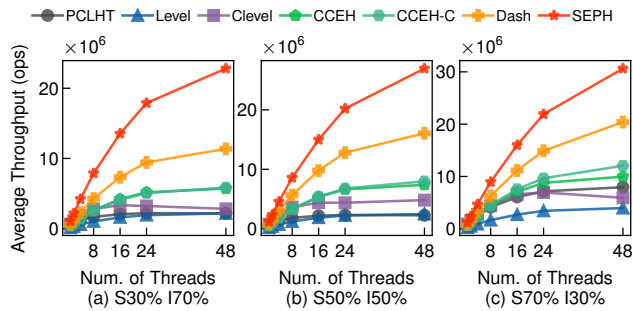


Figure 13: Performance Scalability under YCSB Workloads with Different Search/Insert Ratio (%).

### 4.3 Macro Benchmark

Figure 14 shows the performance results of executing the standard YCSB workloads in terms of the minimal, average, and maximal throughputs. First of all, under the workload Load (100% insertion) shown in Figure 14(a), SEPH performs  $2.63\times \sim 19.59\times$  higher average throughput and at least  $4.52\times$  higher minimal throughput than any other evaluated hashing schemes. This demonstrates SEPH can deliver the most superior performance efficiency and predictability under the insertion-intensive workload, with the help of the proposed low-overhead splitting and semi lock-free concurrency control.

However, under the workload C (100% search) shown in Figure 14(d), the average throughput of SEPH is 12.83% lower than that of Dash, since SEPH is not optimized for search operations (as discussed in §4.2.3). But it is encouraging to see that with the increasing of update operations, the performance gap between the average throughputs of SEPH and Dash reduces and even reverses, because the proposed semi lock-free concurrency control avoids the PM writes to manipulate locks for update operations. Particularly, under the workload B (95% search & 5% update) and the workload F (95% search & 5% read-modify-write (RMW)), the average throughput of SEPH only falls behind that of Dash by 10.9% and 5.6%, as shown in Figures 14(c) and 14(f), respectively. On the contrary, under the workload A (50% search & 50% update), the average throughput of SEPH overtakes that of Dash by 6.9% as in Figure 14(f).

More importantly, under the workload D (95% search & 5% insertion) shown in Figure 14(e), even though SEPH does not achieve the best average throughput (due to the high portion of search operations) among all the evaluated hashing schemes, SEPH demonstrates the best performance predictability (i.e., improving the minimal throughput by at least 39.50%). This reveals the value of the proposed low-overhead splitting in reducing the resizing overhead, even if there are only 5% of insertions. Figure 15 further shows the operation latency of the evaluated hashing schemes at different percentiles under the workload D. It can be observed that SEPH outperforms

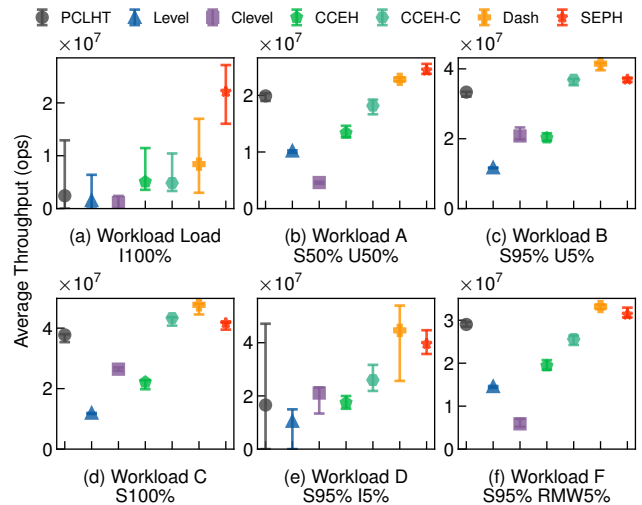


Figure 14: The Minimal, Average, and Maximal Throughputs under Standard YCSB Workloads.

EH-based designs by at least  $11\times$  and  $1.82\times$  for the 99.999th and 100th percentile latency respectively; additionally, SEPH greatly surpasses Level-based designs and PCLHT by at least  $2792\times$  for the 100th percentile latency.

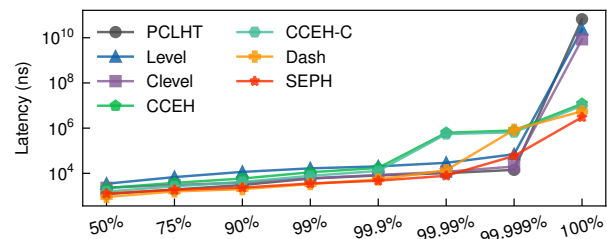


Figure 15: Latency at Different Percentiles (Workload D).

## 5 Conclusion

This paper presents SEPH, a Scalable, Efficient, and Predictable Hashing for PM. To break the dilemma between efficiency and predictability, SEPH introduces a new structure called level segment (LS) to build the hash table with a unique indexing mechanism. SEPH further enables a low-overhead split operation to significantly suppress the resizing-incurred performance unpredictability, and puts forward a semi lock-free concurrency control that requires a nearly-minimal amount of writes to handle an insertion operation for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency. Our results reveal that SEPH achieves higher efficiency, better scalability, and more reliable predictability when compared with state-of-the-art hashing schemes for PM.

## Acknowledgements

We thank our shepherd, Ashvin Goel, and all the anonymous reviewers for their valuable suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project Nos. CUHK14210320 and CUHK14208521).

## References

- [1] Extended asynchronous dram refresh (eadr), <https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html>.
- [2] Intel corporation. intel 64 and ia-32 architectures software developer's manual, <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [3] Intel optane dc persistent memory module, <https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html>, 2019.
- [4] Intel architecture instruction set extensions programming reference, <https://www.intel.com/content/www/us/en/developer/overview.html#gs.jevnd1>, 2021.
- [5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.
- [6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.
- [7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. *Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated*, page 487–498. Association for Computing Machinery, New York, NY, USA, 2011.
- [8] Daniel Bittman, Darrell DE Long, Peter Alvaro, and Ethan L Miller. Optimizing systems for byte-addressable {NVM} by reducing bit flipping. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 17–30, 2019.
- [9] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.
- [10] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.
- [11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- [12] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, New York, NY, USA, 2015. Association for Computing Machinery.
- [13] Carla Schlatter Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116, 1983.
- [14] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 371–384, 2013.
- [15] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, 1998.
- [16] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.
- [17] H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205(2):225–241, 2007.
- [18] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, dec 1992.
- [19] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.
- [20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.
- [21] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splits: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM*



- Symposium on Operating Systems Principles, SOSP '19*, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] Giorgos Kappes and Stergios V. Anastasiadis. A lock-free relaxed concurrent queue for fast work distribution. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '21*, page 454–456, New York, NY, USA, 2021. Association for Computing Machinery.
- [23] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers accelerating index traversals for in-memory databases. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, 2013.
- [24] Per-Ake Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, apr 1988.
- [25] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, February 2017. USENIX Association.
- [26] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.
- [27] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. Loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 263–279, New York, NY, USA, 2021. Association for Computing Machinery.
- [28] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043, 2021.
- [29] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17*, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.
- [30] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6, VLDB '80*, page 212–223. VLDB Endowment, 1980.
- [31] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: Optimizing persistent index performance on 3dxdpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, March 2020.
- [32] Yujie Liu and Michael Spear. A lock-free, array-based priority queue. *ACM SIGPLAN Notices*, 47(8):323–324, 2012.
- [33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(8):1147–1161, April 2020.
- [34] Chen Luo and Michael J. Carey. On performance stability in lsm-based storage systems. *Proc. VLDB Endow.*, 13(4):449–462, dec 2019.
- [35] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. {ROART}: Range-query optimized persistent {ART}. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 1–16, 2021.
- [36] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.
- [37] ORACLE. Architectural overview of the oracle zfs storage appliance, <https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overviewzfsa-2099942.pdf>, 2018.
- [38] Swapnil Patil and Garth Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.
- [39] Yaqiong Peng and Zhiyu Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):843–857, 2018.
- [40] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST '02*, page 19–es, USA, 2002. USENIX Association.

- [41] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, may 2006. 461–476, Carlsbad, CA, October 2018. USENIX Association.
- [42] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.
- [43] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.
- [44] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.
- [45] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.
- [46] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.
- [47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014.
- [48] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.
- [49] Pengfei Zuo and Yu Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2018.
- [50] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages



# No Provisioned Concurrency: Fast RDMA-codedigned Remote Fork for Serverless Computing

Xingda Wei<sup>1,2</sup>, Fangming Lu<sup>1</sup>, Tianxia Wang<sup>1</sup>, Jinyu Gu<sup>1</sup>, Yuhan Yang<sup>1</sup>, Rong Chen<sup>1,2</sup>, and Haibo Chen<sup>1</sup>

<sup>1</sup>Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

<sup>2</sup>Shanghai AI Laboratory

## Abstract

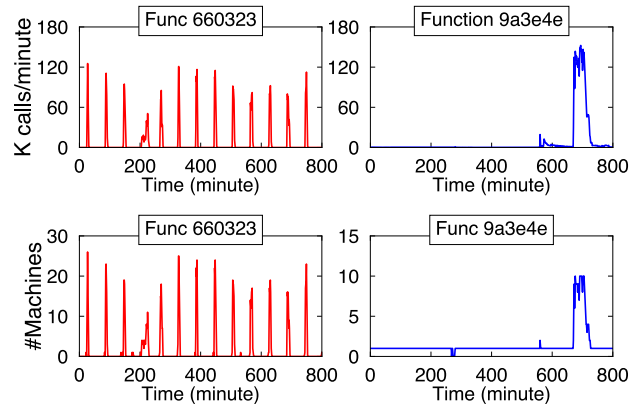
Serverless platforms essentially face a tradeoff between container startup time and provisioned concurrency (i.e., cached instances), which is further exaggerated by the frequent need for remote container initialization. This paper presents MITOSIS, an operating system primitive that provides fast remote fork, which exploits a deep codesign of the OS kernel with RDMA. By leveraging the fast remote read capability of RDMA and partial state transfer across serverless containers, MITOSIS bridges the performance gap between local and remote container initialization. MITOSIS is the first to fork over 10,000 new containers from one instance across multiple machines within a second, while allowing the new containers to efficiently transfer the pre-materialized states of the forked one. We have implemented MITOSIS on Linux and integrated it with FN, a popular serverless platform. Under load spikes in real-world serverless workloads, MITOSIS reduces the function tail latency by 89% with orders of magnitude lower memory usage. For serverless workflow that requires state transfer, MITOSIS improves its execution time by 86%.

## 1 Introduction

Serverless computing is an emerging cloud computing paradigm supported by major cloud providers, including AWS Lambda [23], Azure Functions [91], Google Serverless [44], Alibaba Serverless Application Engine [30] and Huawei Cloud Functions [58]. One of its key promises is *auto-scaling*—users only provide serverless functions, and serverless platforms automatically allocate computing resources (e.g., containers<sup>1</sup>) to execute them. Auto-scaling makes serverless computing economical: the platform only bills when functions are executed (no charge for idle time).

However, *coldstart* (i.e., launching a container from scratch for each function) is a key challenge for fast auto-scaling, as the start time (over 100 ms) can be orders of magnitude higher than the execution time for ephemeral serverless functions [37, 94, 121]. Accelerating coldstart has become a hot topic in both academia and industry [41, 122, 94, 17, 102, 37, 20]. Most of them resort to a form of ‘warmstart’ by *provisioned*

<sup>1</sup>We focus on executing serverless functions with containers in this paper, which is widely adopted by existing platforms [122, 123, 54, 64].



**Figure 1.** The timelines of call frequency (top) and sufficient resource provisioning (bottom) for two serverless functions in a real-world trace from Azure Functions [102].

concurrency, e.g., launching a container from a cached one. However, they require non-trivial resources when scaling functions to a distributed setting, e.g., each machine should deploy many cached containers.

Unfortunately, scaling functions to multiple machines is common because a single machine has a limited function capacity to handle the timely load spikes. Consider the functions sampled from real-world traces of Azure Functions [102]. The request frequency of function 9a3e4e can surge to over 150 K calls per minute, increased by 33,000 $\times$  within one minute (see the top of Figure 1). To avoid stalling numerous newly arriving function calls, the platform should immediately launch sufficient containers across multiple machines (see the bottom part of Figure 1). Due to the unpredictable nature of the serverless workload, it is challenging for the platform to decide the number of cached instances for the warmstart. Hence, there is ‘no free lunch’ for such resources: commercial platforms require users to reserve and pay for them to achieve better performance (i.e., lower response time), e.g., AWS Lambda Provisioned Concurrency [12].

Even worse, dependent functions that run in separate containers cannot directly transfer states. Instead, they must resort to message passing or cloud storage for state transfer, which introduces data serialization/de-serialization, memory copy and storage stack overheads. Recent reports have shown that these may count up to 95% of the function execution



time [71, 53]. Unfortunately, transferring states between functions is common in serverless workflows—a mechanism to compose functions into more complex applications [4, 2]. Though recent research [71] bypasses these overheads for local state transfer (i.e., functions that run on the same machine) by co-locating local functions in the same container, it is still unclear how to do so in a remote setting.

We argue that *remote fork* (forking containers across machines like a local fork) is a promising primitive to enable both efficient function launching and fast function state sharing. First, the fork mechanism has been shown efficient in both performance and resource usage for launching containers on a single machine: one cached container is sufficient to start numerous containers with 1 ms time [17, 37, 36]. By extending the fork mechanism to remote, one active container is sufficient to start numerous containers efficiently on all the machines, achieving *no provisioned concurrency* in a distributed setting. Second, remote fork provides transparent intermediate state sharing between remote functions: the code in the container created by the fork can access the pre-materialized states of the forked container transparently bypassing message passing or cloud storage.

However, state-of-the-art systems can only achieve a conservative remote fork with Checkpoint/Restore techniques (C/R) [7, 117]. Our analysis reveals that they are not efficient for serverless computing, i.e., even slower than coldstart due to the costs of checkpointing the memory of parent container into files, transferring the files through the network and accessing the files through a distributed file system (§3). Even though we have utilized modern interconnects (i.e., RDMA) to reduce these costs, the software overhead of checkpointing and distributed file accesses still make C/R underutilize the low latency and high throughput of RDMA.

We present MITOSIS, an operating system primitive that provides a fast *remote fork* by deeply co-designing with RDMA. The key insight is that the OS can directly access the physical memory on remote machines via RDMA-capable NICs (RNICs) [115], which is extremely fast thanks to bypassing remote OS and remote CPU. Therefore, we can realize remote fork by imitating local fork through mapping a child container’s virtual memory to its parent container’s physical memory without checkpointing the memory. The child container can directly read the parent memory in a copy-on-write fashion using RNIC, bypassing the software stacks (e.g., distributed file system) introduced by traditional C/R.

Leveraging RDMA for remote fork with kernel poses several new challenges (§4.1): (1) fast and scalable RDMA-capable connection establishment, (2) efficient access control of the parent container’s physical memory and (3) efficient parent container lifecycle management at scale. MITOSIS addresses these challenges by (1) retrofitting advanced RDMA feature (i.e., DCT [1]), (2) proposing a new connection-based memory access control method designed for remote fork and (3) co-designing container lifecycle management with the

help of serverless platform. We also introduce techniques including generalized lean container [94] to reduce containerization overhead for the remote fork. In summary, we show that remote fork can be made efficient, feasible and practical on commodity RNICs for serverless computing.

We implemented MITOSIS on Linux with its core functionalities written in Rust as a loadable kernel module. It can remote-fork 10,000 containers on 5 machines within 0.86 second. MITOSIS is fully compatible with mainstream containers (e.g., runC [13]), making integration with existing container-based serverless platforms seamlessly. To demonstrate the efficiency and efficacy, we integrated MITOSIS with Fn [123], a popular open-source serverless platform. Under load spikes in real-world serverless workloads, MITOSIS reduces the 99<sup>th</sup> percentile latency of the spiked function by 89% with orders of magnitude lower memory usage. For a real-world serverless workflow (i.e., FINRA [14]) that requires state transfer, MITOSIS reduces its execution time by 86%.

**Contributions.** We highlight the contributions as follows:

- **Problem:** An analysis of the performance-resource provisioning trade-off of existing container startup techniques, and the costs of state transfer between functions (§2).
- **MITOSIS:** An RDMA-co-designed OS remote fork that quickly launches containers on remote machines without provisioned concurrency and enables efficient function state transfer (§4–5).
- **Demonstration:** An implementation on Linux integrated with Fn (§6) and evaluations on both microbenchmarks and real-world serverless applications demonstrate the efficacy of MITOSIS (§7). MITOSIS is publicly available at <https://github.com/ProjectMitosOS>.

## 2 Background and Motivation

### 2.1 Serverless computing and container

Serverless computing is a popular programming paradigm. It abstracts resource management from the developers: they only need to write the application as *functions* in a popular programming language (e.g., Python), upload these *functions* (as container images) to the platform, and specify how to call them. The platform can *auto-scale* according to function requests by dynamically spawning a container [54, 123, 59, 94, 22, 30, 91, 44, 22, 70]<sup>2</sup> to handle each call. The spawned containers will also be automatically reclaimed after functions return, making serverless economical: the developers only pay for the in-used containers.

Container is a popular host for executing functions. It not only packages the application’s dependencies into a single image that eases the function deployment, but also provides lightweight isolation through Linux’s `cgroups` and `namespaces`, which is necessary to run applications in a multi-tenancy environment. Unfortunately, enabling container

<sup>2</sup>Serverless platform may use virtual machines to run functions, which is not the focus of this paper.

**Table 1:** A comparison of startup techniques for autoscaling  $n$  concurrent invocations of one function to  $m$  machines. Local means the resources for the startup are provisioned on the function execution machine. The function is a simple python program that prints ‘hello world’.

	<b>Coldstart</b> [9, 119]	<b>Caching</b> [63, 123, 94, 102, 122]	<b>Fork</b> [37, 17, 36]	<b>Checkpoint/Restore</b> [120, 37, 117, 20]	<b>Remote fork</b> MITOSIS
<b>Local startup performance</b>	Very slow (100 ms)	Very fast (< 1 ms)	Fast (1 ms)	Medium (5 ms)	Fast (1 ms)
<b>Remote startup performance</b>	Very slow (1,000 ms)	N/A	N/A	Slow (24 ms)	Fast (3 ms)
<b>Overall resource provisioning</b>	$O(1)$	$O(n)$	$O(m)$	$O(1)$	$O(1)$

introduces additional function startup costs and state transferring costs due to container bootstrap and segregated function address spaces, respectively.

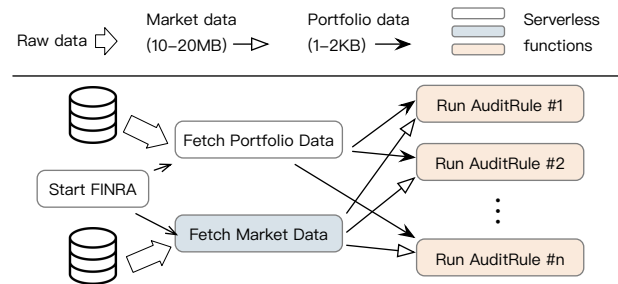
## 2.2 Startup and resource provisioning costs

**Coldstart performance cost.** Starting a container from scratch, commonly named as ‘coldstart’, is notoriously slow. The startup includes pulling the container image, setting up the container configurations and initializing the function language runtime. All the above steps are costly, which take even more than hundreds of milliseconds [37, 94]. As a result, coldstart may dominate the end-to-end latency of ephemeral serverless functions [37, 94, 119, 33]. For example, Lambda@Edge reports that 67% of its functions run in less than 20 ms [33]. In comparison, starting a Hello-world python container with runC [13]—a state-of-the-art container runtime—takes 167 ms and 1783 ms when the container image is stored locally and remotely, respectively (see Table 1).

**Warmstart resource cost due to provisioned concurrency.** A wealth of researches focus on reducing the startup time of coldstart with ‘warmstart’ techniques [94, 17, 37, 102, 113, 42, 119, 131, 106]. However, they must pay more resource provisioning cost (see Table 1):

**Caching** [63, 64, 123, 41, 122, 94, 17, 102]. By caching finished containers (e.g., via Docker pause [8]) instead of reclaiming them, future functions can reuse cached ones (e.g., via Docker unpause) with nearly no startup cost (less than 1 ms). However, Caching consumes large in-memory resources: the resource provisioned—number of the cached instances ( $O(n)$ ) should match the number of concurrent functions ( $n$ ), because a paused container can only unpause for one function. Given the unpredictability of the number of function invocations (e.g., load spikes in Figure 1), it is challenging for the developers or the platform to decide how many cached instances are required. Thus, Caching inevitably faces the trade-off between fast startup and low resource provisioning, resulting in huge cache misses.

**Fork** [37, 17, 36]. A cached container (*parent*) can call the fork system call (instead of unpause) to start new containers (*children*). Since fork can be called multiple times, each machine only requires one cached instance to fork new containers. Thus, fork reduces resource provisioned of Caching—cached containers from  $O(n)$  to  $O(m)$ , where  $m$  is the number of machines that require function startup. However, it is



**Figure 2.** The workflow graph of a real-world serverless application, Financial Industry Regulatory Authority, FINRA [14].

still proportional to the number of machines ( $m$ ) since fork cannot generalize to a distributed setting.

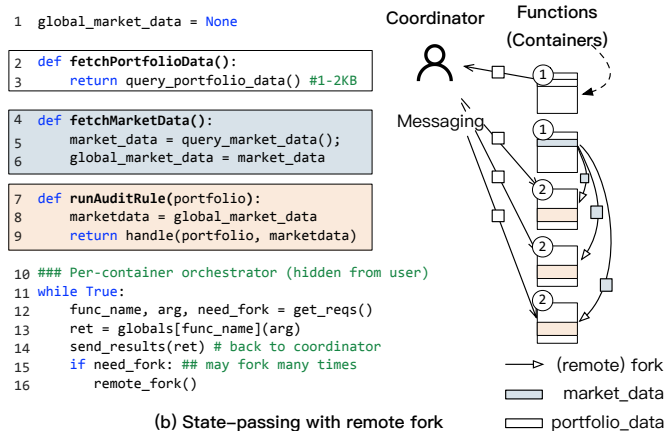
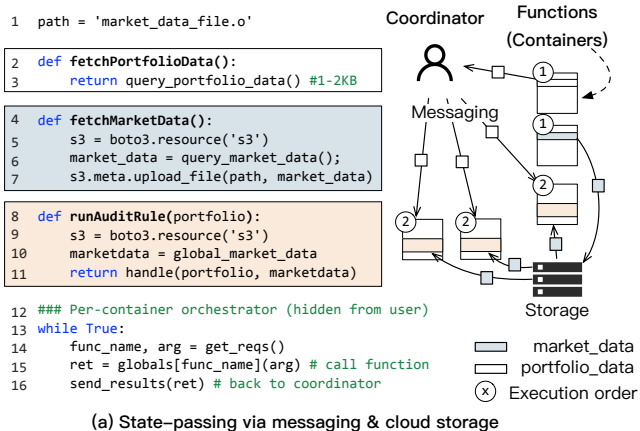
**Checkpoint/Restore (C/R)** [120, 37, 117]. C/R starts containers from container checkpoints stored in a file. It only needs  $O(1)$  resource (the file) to warmstart, because the file can be transferred through the network if necessary. Though being optimal in resource usage, C/R is orders of magnitude slower than Caching and fork. We analyze it in §3 in detail.

## 2.3 (Remote) state transfer cost

Transferring states between functions is common in serverless workflows [36, 17, 95, 64, 4, 2]. A workflow is a graph describing the producer-consumer relationships between functions. Consider the real-world example FINRA [14] shown in Figure 2. It is a financial application that validates trades according to the trade (Portfolio) and market (Market) data. Upstream functions (the ones that produce states, i.e., `fetchPortfolioData` and `fetchMarketData` first read data from external sources. Afterward, they transfer the results to many downstream functions (the one that consumes states), i.e., `runAuditRules` to process them concurrently for a better performance.

Functions run in different containers can only transfer states either by copying them through the network via message passing or exchanging them at a cloud storage service. Figure 3 (a) shows a simplified code for running FINRA on AWS Lambda. For small states transfers (less than 32KB, e.g., Portfolio), Lambda piggybacks the states in messages exchanged between the coordinator and the function containers [131]. For large ones (Market), functions must exchange them with S3—Lambda’s cloud storage service.

Transferring states via messages and cloud storage inevitably faces the overheads of data serialization, memory



**Figure 3.** (a) A simplified code of FINRA (see Figure 2) on existing serverless platforms. (b) A simplified code of using (remote) fork to transfer states between FINRA functions. `globals` records a mapping between function name and its pointer.

copies, and cloud storage stacks, causing up to a 1,000X slow-down [53, 71]. To cope with the issue, existing work proposes serverless-optimized messaging primitives [17] or specialized storage systems [110, 69, 96], but none of the mentioned overhead is completely eliminated [71]. Faastlane [71] co-locates functions in the same container with *threads* so that it can bypass these overheads with shared memory accesses. However, threads cannot generalize to a distributed setting. Faastlane fallbacks to message passing if the upstream and downstream functions are on different machines. SPRIGHT [97] achieves a similar effect by retrofitting eBPF. However, they don’t support efficient data sharing across nodes.

### 3 Remote Fork for Serverless Computing

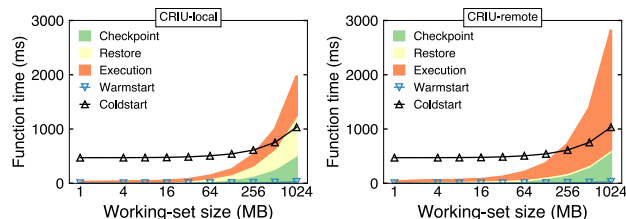
We show the following two benefits of *remote fork* to address the issues mentioned in the previous section.

**Efficient (remote) function launching.** When generalizing the FORK primitive to a remote setting, a single *parent* container is sufficient to launch subsequent *child*<sup>3</sup> containers across the cluster, similar to C/R (see Table 1). We believe  $O(1)$  resource provisioning is desirable for the developers/tenants since they only need to specify whether they need resource for warmstart, instead of how many (e.g., the number of machines for forking or cached instances [12] for Caching).

**Fast and transparent (remote) state transfer.** The FORK primitive essentially bridges the address spaces of parent and child containers. The transferred states are pre-materialized in the parent memory, so the child can seamlessly access them with shared memory abstraction with no data serialization, zero-copy (for read-only accesses<sup>4</sup>) and cloud storage costs. Meanwhile, the *copy-on-write* semantic in the FORK primitive avoids the costly memory coherence protocol in traditional distributed shared memory systems [75, 57].

<sup>3</sup>We may also call the kernel/machine hosting the parent/child container as *parent/child* in this paper without losing generality.

<sup>4</sup>In the case of the traditional fork. MITOSIS further optimizes with one-sided RDMA (§4), allowing zero-copy even for read-write accesses.

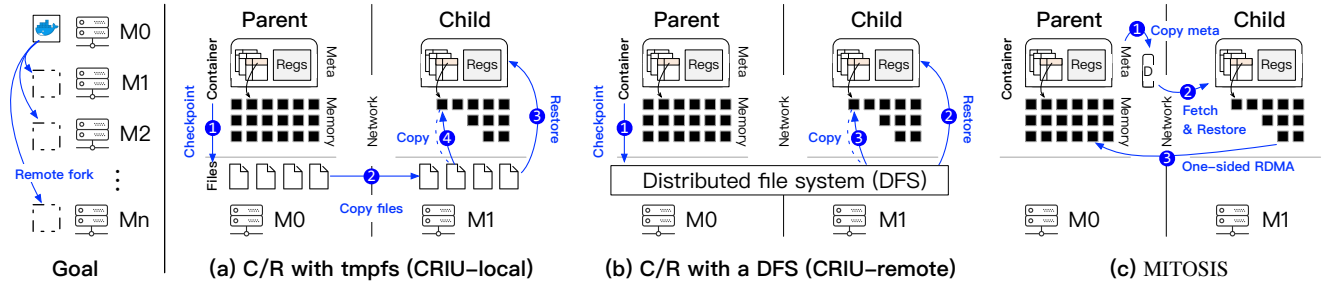


**Figure 4.** Analysis of using C/R for remote fork. **Setup:** CRIU-local: CRIU with a local file system (e.g., *tmpfs*), which uses RDMA to transfer files between machines. CRIU-remote: CRIU with an RDMA-accelerated distributed file system (e.g., *Ceph* [89]).

Figure 3 (b) presents a concrete example of using fork to transfer market data in FINRA (see Figure 2). In this setup, all functions are packaged in the same container, which has an orchestrator dispatching function requests to user-implemented functions (lines 11–14).<sup>5</sup> We further assume the coordinator issuing requests to the orchestrators is fork-aware (§6.1): based on the function dependencies in the workflow graph (e.g., Figure 2), it will request the orchestrator to fork children if necessary (line 12). After the orchestrator finishes `fetchMarketData` (line 13), it forks (lines 15–16) to run downstream functions (`runAuditRule`), which can directly access the `global_market_data` pre-materialized by the parent (line 8).

**Challenge: remote fork efficiency.** To the best of our knowledge, existing containers can only remote fork with a C/R-based approach [108, 32]. To fork a child, the parent first *checkpoints* its states (e.g., register values and memory pages) by copying them to a file, and then *transfers* the file to the child—either using a remote file copy—see *CRIU-local* in Figure 5 (a), or a distributed file system (see *CRIU-remote* in Figure 5 (b)). After receiving the file, the child *restores* the parent’s execution by loading the container states from the checkpointed file. Note that C/R may load some states (i.e., memory pages) on-demand for better performance [120].

<sup>5</sup>This setup is common in serverless platforms [70, 71, 2].



**Figure 5.** An overview of different approaches to achieve ultra-fast remote fork, including (a) C/R with a local filesystem (e.g., tmpfs), (b) C/R with a fast distributed filesystem (e.g., Ceph [5]), and (c) MITOSIS.

Unfortunately, the C/R-based remote fork is not efficient enough for serverless computing. Figure 4 (a) shows the execution time of serverless functions on a remote machine using CRIU [7]—the state-of-the-art C/R on Linux (with careful optimizations, see §7 for details) to realize CRIU-local and CRIU-remote. The synthetic function randomly touches the entire parent’s memory. We observe that C/R-based remote fork can even be  $2.7 \times$  slower than coldstart if it accesses 1 GB remote memory. We attribute it to one or more of the following aspects.

**Checkpoint container memory.** CRIU takes 9 ms (resp. 518 ms) and 15.5 ms (resp. 590 ms) to checkpoint 1 MB (resp. 1 GB) memory of the parent container using local or distributed file systems, respectively. The overhead is dominated by copying the memory to the files: unlike the local fork, the child’s OS resides on another machine and thus, lacks direct memory access capability to the parent’s memory pages.

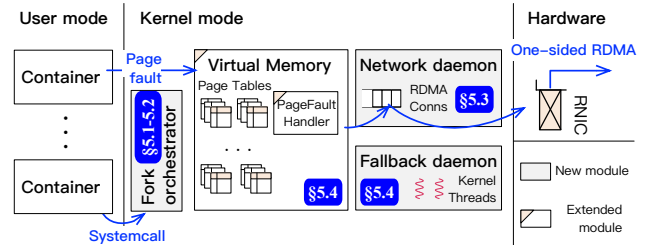
**Copy checkpointed file.** For CRIU-local, transferring the entire file from the parent to the child takes 11–734 ms for 1 MB–1 GB image (compared to the 0.61–570 ms execution time), respectively. The whole file copy is typically unnecessary since serverless functions typically access a partial state of the parent container [120] (see also Figure 16 (b)).

**Additional restore software overhead.** CRIU-remote enables on-demand file transfer<sup>6</sup>: it only reads the required remote file pages during page faults. However, the execution time is  $1.3\text{--}3.1 \times$  longer than CRIU-local because each page fault requires a DFS request to read the page: the DFS latency (100  $\mu$ s) is much higher than local file accesses. More importantly, the latency is much higher than one network round-trip time (3  $\mu$ s) due to the software overhead.

## 4 The MITOSIS Operating System Primitive

**Opportunity: kernel-space RDMA.** Remote Direct Memory Access (RDMA) is a fast networking feature widely deployed in data-centers [115, 47, 43]. Though commonly used in the user-space, RDMA further gives the kernel the ability to read/write the *physical memory* of remote machines [115]

<sup>6</sup>CRIU lazy migration [6] also supports on-demand transfer. However, it is not optimized for RDMA and is orders of magnitude slower than our evaluated CRIU-remote (210 vs. 42 ms) for the python hello function.



**Figure 6.** The MITOSIS architecture.

bypassing remote CPUs (i.e., one-sided RDMA READ), with low latency (e.g., 2  $\mu$ s) and high bandwidth (400 Gbps).

**Approach: imitate fork with RDMA.** MITOSIS achieves an efficient remote fork by imitating the local fork with RDMA. Figure 5 (c) shows an overview. First, we copy the parent’s metadata (e.g., page table) to a condensed descriptor (§5.1) to fork a child (1). Note that unlike C/R, we don’t copy the parent’s memory pages to the descriptor. The descriptor is then copied to the child via RDMA to recover the parent’s metadata, similar to `copy_process` in the local fork (2). During execution, we configure the child’s remote memory accesses to trigger page faults, and the kernel will read the remote pages accordingly. The fault handler is triggered naturally in an on-demand pattern, which avoids transferring the entire container state. Meanwhile, MITOSIS directly uses one-sided RDMA READ to read the remote physical memory (3), bypassing all the software overheads.

**Architecture.** We target a decentralized architecture—each machine can fork from others and vice versa. Note that we do not require dedicated resources (e.g., pinned memory) to fork containers, thus, non-serverless applications can co-run with MITOSIS. We realize MITOSIS by adding four components to the kernel (see Figure 6): The *fork orchestrator* rehearsals the remote fork execution (§5.1 and 5.2). The *network daemon* manages a scalable RDMA connection pool (§5.3) for communicating between kernels. We extend OS’s *virtual memory subsystems* to utilize the remote memory with RDMA (§5.4). Finally, *fallback daemon* provides RPC handlers to restore rare remote memory accesses that cannot utilize RDMA.

**Security model.** We preserve the security model of containers, i.e., the OS and hardware (RNIC) are trustworthy while malicious containers (functions) may exist.



## 4.1 Challenges and approaches

**Efficient and scalable RDMA connection setup.** Though RDMA is fast (e.g.,  $2\ \mu\text{s}$ ), it is traditionally only supported in the connection-oriented transport (RC) [35, 83, 126, 125, 105, 127, 124], where connection establishment is much slower (e.g., 4 ms [11] with a limited 700 connections/second throughput). Caching connections to other machines can mitigate the issue, but it is impractical when RDMA-capable clusters have scaled to more than 10,000 nodes [43].

We retrofit DCT [1], an underutilized but widely supported advanced RDMA feature with fast and scalable connection setups to carry out communications between kernels (§5.3).

**Efficient remote physical memory control.** MITOSIS exposes the parent’s physical memory to the children for the fastest remote fork. However, this approach introduces consistency problems in corner cases. If the OS changes a parent’s virtual–physical mappings [77, 80, 78, 79] (e.g., swap [78]), the children will read an incorrect page. User-space RDMA can use memory registration (MR) [93] for the access control. However, MR has non-trivial registration overheads [49]. Further, kernel-space RDMA has limited support for MR—it only supports MR on RCQP (with FRMR [90]).

We propose a registration-free memory control method (§5.4) that transforms RNIC’s memory checks to connection permission checks. We further make the checks efficient by utilizing DCT’s scalable connection setup feature.

**Parent container lifecycle management.** For correctness, we must ensure a forked container (parent) is alive until all its successors (including children forked from the children) finish. A naive approach is letting each machine track the lifecycles of the successors of its hosting parents. However, it would pose significant management burdens: a parent’s successors may span multiple machines, forming a distributed *fork tree*. Meanwhile, each machine may have multiple trees. Consequently, each machine needs extensive communications with the others following paths in the trees to ensure a parent can be safely reclaimed.

To this end, we onload the lifecycle management to the serverless platform (§6.3). The observation is that serverless coordinators (nodes that invoke functions via fork) naturally maintain the runtime information of the forked containers. Thus, they can trivially decide when to reclaim parents.

## 5 Design and Implementation

For simplicity, we first assume one-hop fork (i.e., no cascading) and then extend to multi-hops fork (see §5.5).

**API.** We decouple the fork into two phases (see Figure 7): The user can first call `fork_prepare` to generate the parent’s metadata (called *descriptor*) related to remote fork. The descriptor is globally identified by the local unique `handle_id` and `key` (generated and returned by the prepared call) and the parent machine’s RDMA address. Given the identifier,

```
// Prepare the container descriptor at the parent machine
status_t fork_prepare(uint64_t *handle_id, uint64_t *key);

// Resume from a parent descriptor at the child machine
status_t fork_resume(char *addr, uint64_t handle_id, uint64_t key);
```

Figure 7. The major MITOSIS remote fork system calls.

users can start a child via `fork_resume` at another machine (can be the same as the parent, i.e., local fork).

Compared to the traditional one-stage fork system call, a two-phase fork API (prepare and resume)—similar to `pause` and `unpause` in Caching is more flexible for serverless computing. For example, after preparing and recording the parent’s identifier at the coordinator, it can later start children without communicating with the parent machine.

**Visibility of the parent’s data structures.** By default, MITOSIS exposes all the parent’s data structures—including virtual memory and file descriptors, to the child after `fork_prepare`. MITOSIS could introduce APIs to let the application limit the scope of the exposure, but currently, we find it unnecessary: parents must trust the children since they are from the same application.

### 5.1 Fork prepare

`fork_prepare` will generate a local in-memory data structure (*container descriptor*) capturing the parent states, which contains (1) cgroup configurations and namespace flags—for containerization, (2) CPU register values—for recovering the execution states, (3) page table and virtual memory areas (VMAs)—for restoring the virtual memory, and (4) opened file information—for recovering the I/O. We follow local fork (e.g., Linux’s `copy_process()`) to capture (1)–(3) and CRIU [7] for (4). Since deciding when to reclaim a descriptor is challenging, we always keep the prepared parents (and their descriptors) alive unless the serverless platform explicitly frees them (i.e., via `fork_reclaim`).

Though the descriptor plays a similar role as C/R checkpointed file, we emphasize one key difference: the descriptor only stores the page table, not the memory pages. As a result, it is orders of magnitude smaller (KB vs. MB) and orders of magnitude faster to generate and transfer.

### 5.2 Fork resume

`fork_resume` resumes the parent’s execution state by fetching the parent descriptor and then restoring from it. We now describe how to make the above two steps fast. For now, we assume the child OS has established network connections capable of issuing RPCs and one-sided RDMA to the parent. The next section describes the connection setup.

**Fast descriptor fetch with one-sided RDMA.** A straightforward implementation of fetching the descriptor is using RPC. However, RPC’s memory copy overhead is non-trivial (see Figure 18), as the descriptor of a moderate-sized container may consume several KBs. The ideal fetch is using one one-sided RDMA READ, which requires (1) storing the parent’s

descriptor into a consecutive memory area and (2) informing the child’s OS of the memory’s address and size in advance.

The first requirement can be trivially achieved by serializing the descriptor into a well-format message. Data serialization has little cost (sub-millisecond) due to the simple data structure of descriptor. For the second requirement, a naive solution is to encode the memory information in the descriptor identifier (e.g., `handler_id`) that is directly passed to the resume system call. However, this approach is insecure because a malicious user could pass a malformed ID, causing the child to read and use a malformed descriptor. We adopt a simple solution to remedy this: MITOSIS will send an authentication RPC to query the descriptor memory information with the descriptor identifier. If the authentication passes, the parent will send back the descriptor’s stored address and payload so that the child can directly read it with one-sided RDMA. We chose a simple design because the overhead of an additional RPC (several bytes) is typically negligible: reading the descriptor (several KBs) will dominate the fetch time.

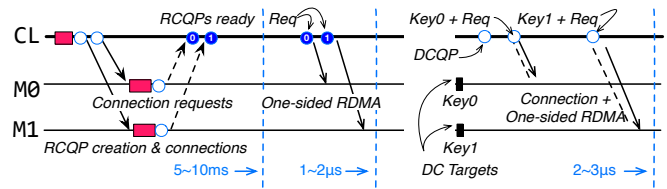
**Fast restore with generalized lean containers.** With the fetched descriptor, child OS uses the following two steps to resume a child to the parent’s execution states: (1) Containerization: set the `cgroups` and `namespaces` to match the parent’s setup; (2) Switch: replace the caller’s CPU registers, page table, and I/O descriptors with the parent’s. The switch is efficient (finishes in sub-milliseconds): it just imitates the local fork—e.g., unmapping the caller’s current memory mapping and mapping the child’s virtual memory to the parents by copying parent’s page table to the child. On the other hand, containerization can take tens of milliseconds due to the cost of setting `cgroups` and `namespaces`.

Fortunately, fast containerization has been well-studied [94, 17, 27, 112]. For instance, SOCK [94] introduces *lean container*, which is a special container having the minimal configurations necessary for serverless computing. It further uses pooling to hide the cost of container bootstrap, reducing its time from tens of milliseconds to a few milliseconds. We generalize SOCK’s lean container to a distributed setting to accelerate the containerization of the remote fork. Specifically, before resuming a remote parent, we will use SOCK to create an empty lean container that satisfies the parent’s isolation requirements. Afterward, the empty container calls MITOSIS to resume execution. Since the container has been properly configured with SOCK, we can skip the costly containerization.

### 5.3 Network daemon

The network daemon aims to reduce the costs of creating RDMA connections (commonly called *RCQP*) on the critical path of the remote fork. Meanwhile, it also avoids caching RCQPs connected to all the servers to save memory.

**Solution: Retrofit advanced RDMA transport (DCT).** The essential requirement behind the goal is that we need QP to be connectionless. RDMA does provide a connectionless



**Figure 8.** A comparison of a client (CL) using two RCQPs and DCQP to communicate with two machines (M1 and M2).

transport—unreliable datagram (UD), but it only supports messaging, so we can just use it for RPC.

We find dynamic connected transport (DCT) [1]—a less studied but widely supported RDMA feature suits remote fork well. DCT preserves the functionality of RC and further provides a connectionless illusion: a single DCQP can communicate with different nodes. The target node only needs to create a *DC target*, which is identified by the node’s RDMA address and a 12B *DC key*.<sup>7</sup> After knowing the keys, a child node can send one-sided RDMA requests to the corresponding targets without connection—the hardware will piggyback the connection with data processing and is extremely fast (within  $1\mu\text{s}$  [11, 67]), as shown in Figure 8.

Based on DCT, the network daemon manages a small kernel-space DCQP pool for handling RDMA requests from children. Typically, one DCQP per-CPU is sufficient to utilize RDMA [11]. However, using DCT alone is insufficient because the child needs to know the DCT key in advance to communicate with the parent. Therefore, we also implement a kernel-space FaSST RPC [67] to bootstrap DCT. FaSST is a UD-based RPC that supports connectionless. With RPC, we piggyback the DCT key associated with the parent in the RPC request to query the parent’s descriptor. To save CPU resources, we only deploy two kernel threads to handle RPCs, which is sufficient for our workloads (see Figure 13 (b)).

**Discussion on DCT overheads.** DCT has known performance issue due to extra reconnection messages. Compared with RC, it causes up to 55.3% performance degradations for small (32B) one-sided RDMA READs [67]. Nevertheless, the reconnection has no effect on the large (e.g., more than 1 KB) transfer because transferring data dominates the time [11]. Since the workload pattern of MITOSIS is dominated by large transfers, e.g., reading remote pages in 4KB granularity, we empirically found no influence from this issue.

### 5.4 RDMA-Aware virtual memory management

For resume efficiency, we directly set the page table entries (PTE) of the children’s mapped pages to the parent’s physical addresses (PA) during the resume phase. However, the original OS is unaware of the remote PA in the PTE. Thus, we dedicate a remote bit in the PTE for distinction. In particular, the OS will set the remote bit to be 1 and clear the present bit of the PTE during the switch process at the resume phase. Afterward, child’s remote page access will trap in the kernel

<sup>7</sup>The key consists of a 4 B NIC-generated number and 8 B user-passed key.

**Table 2:** A summary of page fault handling related to remote fork at the child categorized by whether the virtual address (VA) is mapped to remote and whether the remote physical address (PA) is stored.

Example	VA mapped	Parent PA in PTE	Method
Stack grows	No	No	Local
Code in .text	Yes	Yes	RDMA
Mapped file	Yes	No	RPC

after the switch. Consequently, MITOSIS can handle them in the RDMA-aware page fault handler. Note that we don't change the table entry data structure: we utilize an ignored PTE bit (i.e., one in [58 : 52] [60]) for the remote bit.

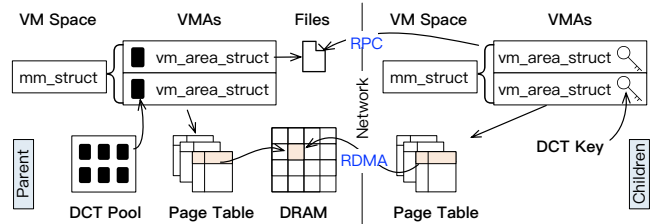
**RDMA-aware page fault handler.** Table 2 summarizes how we handle different faults related to remote fork. If the fault page has not mapped to the parent, e.g., stack grows, we handle it locally like a normal page fault. Otherwise, we check whether the fault virtual address (VA) has a mapped remote PA. If so, we use one-sided RDMA to read the remote page to a local page. Most child pages can be restored via RDMA because serverless function typically touches a subset of the previous run [120, 37]. In case of a missed mapping, we fallback to RPC.

**Fallback daemon.** Each node hosts a fallback daemon that spawns kernel threads to handle children's paging requests, which contains the parent identifier and the requested virtual address. The fallback logic is simple: After checking the validity of the request, the daemon thread will load the page on behalf of the parent. If the load succeeds, we will send the result back to the child.

**Connection-based memory access control and isolation.** Direct exposing the parent's physical memory improves the remote fork speed. Nevertheless, we need to reject accesses to mapped pages that no longer belong to a parent and properly isolate accesses to different containers. Since we expose the memory via one-sided RDMA in a CPU-bypassing way, we can only leverage RNIC for the control.

MITOSIS proposes a connection-based memory access control method. Specifically, we assign different RDMA connections to different portions of the parent's virtual memory area (VMA), e.g., one connection per VMA. If a mapped physical page no longer belongs to a parent, we will destroy the connection related to the page's VMA. Consequently, the child's access to the page will be rejected by the RNIC. The connections are all managed in the kernel to prevent malicious users from accessing the wrong remote container memory.

To make connection-based access control practical, each connection must be efficient in creation and storage. Fortunately, the DCQP satisfies these requirements well. At the child-side, each connection (DC key) only consumes 12B—different DC connections can share the same DCQP. Meanwhile, the parent-side DC target consumes 144B. Note that creating DCQPs and targets also has overheads. Yet, they are



**Figure 9.** An illustration of the extended virtual memory subsystems to map children's virtual addresses to remote memory. The memory space is divided into a list of virtual memory area (VMA)s, each managed by a `vm_area_struct`. DC target pool and DCT keys are used by connection-based memory access control.

logically independent of the parent's memory. Therefore, we use pooling to amortize their creation time (several ms).

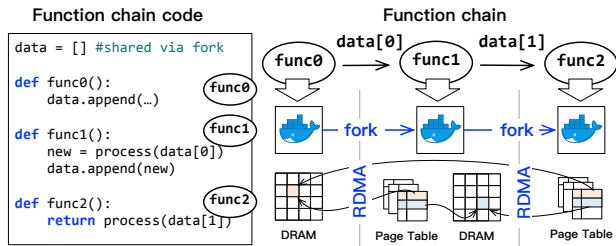
Figure 9 shows the DCT-based access control in action. Upon fork preparation, MITOSIS assigns one DC target—selected from a target pool—to each parent VMA. The pool is initialized during boot time and is periodically filled in the background. The DC keys of these targets are piggybacked in the parent's descriptor so that the children can record them in their VMA during resume. Upon reading a parent's page, the child will use the key corresponding to the page's VMA to issue the RDMA request. With this scheme, if the parent wants to reject accesses to this page, it can destroy the corresponding DC target.

Connection-based control has false positives: after destroying a VMA's assigned target, all remote accesses to it are rejected. Assigning DC targets in a more fine-grained way (e.g., multiple targets per VMA) can mitigate the issue at the cost of increased memory usage. We found it is unnecessary because VA-PA changes are rare at the parent. For example, swap never happens if the OS has sufficient memory.

**Security analysis.** Compared with normal containers, MITOSIS additionally exposes its physical memory to remote machines via RDMA. Nevertheless, since remote containers must leverage their kernels to read the exposed memory, a malicious container cannot read others states as long as its kernel is not compromised. Besides this, the inherent security issues of RDMA [111, 99, 128] may also endanger MITOSIS. While such security threats are out of the scope of our work, it is possible to integrate orthogonal solutions [111, 99, 128, 115] to improve the security of MITOSIS.

**Optimizations: prefetching and caching.** Even with RDMA, reading remote pages is still much slower than local memory accesses [35] (3  $\mu$ s vs. 100 ns). Thus, we apply two standard optimizations: *Prefetching* prefetches adjacent remote pages upon page faults. Empirically, we found a prefetch size of one is sufficient to improve the performance of remote fork at a small cost to the runtime memory (see Figure 15). Thus, MITOSIS only prefetches one adjacent page by default. *Caching* caches the finished children's page table (and the read pages) in the kernel. A later child forking the same parent can then reuse the page table in a copy-on-write way to avoid reading





**Figure 10.** An illustration of multi-hops remote fork.

the touched pages again. This is essentially a combination of local-remote fork. To avoid extra memory cost, we only keep the cached page table for a short period (usually several seconds) to cope with load spikes (e.g., see Figure 1).

### 5.5 Supporting multi-hops remote fork

MITOSIS supports multi-hops fork: a child can be forked again with its children possibly on the third machine. It is similar to one-hop fork except that we need to further track the ownership of remote pages in a fine-grained way. As shown in Figure 10, the pages behind `data[1]` and `data[0]` resides on two different machines. A naive approach would be maintaining a map to track the owner of each virtual page. However, it would consume non-trivial storage overhead. To reduce memory usage, MITOSIS encodes the owner in the PTE: we dedicate 4 bits in the PTE’s ignored bits to encode the remote page machine—supporting a maximum of 15-hops remote fork (up to 15 ancestors)

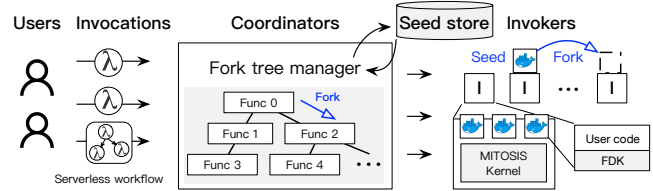
## 6 Bringing MITOSIS to Serverless Computing

This section describes how we apply MITOSIS to FN [123]—a popular open source serverless platform. Though we focus on FN, we believe our methodology can also apply to other serverless platforms (e.g., OpenWhisk [122]) because they follow a similar system architecture (see Figure 11).

**Basic FN.** Figure 11 shows an overview of FN. It handles the function request that is either an invocation of a single function, or an execution of a serverless workflow (e.g., see Figure 2). A dedicated *coordinator* is responsible for scheduling the executions of these requests. The function code must be packed to a container and uploaded to a Docker registry [34] managed by the platform.

To handle the invocation of a single function, the coordinator will direct the request to an *invoker* chosen from a pool of servers. After receiving the request, the invoker spawns a container with Caching to accelerate startups to execute the function. Note that FN hides the mapping of request to user-function (e.g., 12–16 in Figure 3 (a)) with function development kit (FDK): i.e., the user only needs to provide the code for the function, not the code that dispatches the requests to the function. Thanks to this abstraction, we can extend FDK to add the fork capabilities.

To execute a workflow, the coordinator will first decompose the workflow into single-function calls (one



**Figure 11.** Integrating MITOSIS to FN. The gray boxes are our added (or extended) components.

for each workflow graph node), then schedule them based on the dependency relationship. In particular, the coordinator will only execute a downstream function (e.g., `defrunAuditRule` in Figure 2) after all its upstream functions (`fetchPortfolioData` and `fetchMarketData`) finish.

### 6.1 Fork-aware serverless platform

Being aware of MITOSIS, the platform can leverage parents that have prepared themselves via `fork_prepare` (we term them as *seeds* in this paper) to accelerate function startup and state transfer. Besides, it is also responsible for reclaiming the seeds. Based on the use cases, we further categorize seeds into two classes. 1) For seeds that are used for boosting function startups, the frequency of reclamation is low. Hence, we name them *long-lived* seeds and use a coarse-grained reclamation scheme (§6.2). 2) For seeds that are used for state transfer, they only live during the lifecycle of a serverless workflow. We name them *short-lived* seeds and use a fine-grained fork tree-based mechanism to free them (§6.3).

The steps to accelerate FN with MITOSIS are: (1) Extend the FN coordinator to send `prepare/resume` requests to the invoker to fork containers if necessary and (2) Instrument FDK so that it can recognize the new (fork) requests from the coordinator (e.g., line 12–16 in Figure 3 (b)). Since the extensions to the FDK are trivial, we focus on describing the extensions to the coordinator.

**Fork-aware coordinator.** For a single function call, the coordinator first looks up an available (long-lived) seed. The locations of seeds are stored at a *seed store*. If one seed is available, it sends a `fork resume` request to the invoker. Otherwise, we fallback to the vanilla function startup mechanism.

During workflow execution, the coordinator dynamically creates short-lived based on state transfer relationship. Specifically, it will tell the invoker to call `fork_prepare` if it executes an upstream function in the workflow. The prepared results are piggybacked in the reply of the function. Afterward, the coordinator can use `fork_resume` to start downstream functions, which transparently inherit the pre-materialized results of the upstream one.

Note that one function may have multiple upstream functions (e.g., `run AuditRule` in Figure 2). For such cases, we require the user to specify which function to fork by annotating the workflow graph or fuse the upstream functions.



## 6.2 Long-lived seed management

**Deployment.** We deploy long-lived seeds as cached containers because they naturally load the function’s working set into the memory. If the invoker decides to cache a container, it will call `fork_prepare` to generate a seed. Note that we must also adjust FN’s cache policy to be fork-aware. For example, FN always caches a container if it experiences a coldstart, which is unnecessary considering MITOSIS because the fork can accelerate startups more resource-efficiently. Therefore, we only cache the first container facing coldstart across the platform. Moreover, we also detect whether a container is a multi-hop one, i.e., forked from a long-lived seed. We don’t cache such containers as they are short-lived seeds.

**Seed store.** To find the seed information, we record a mapping between function name and the corresponding seed’s RDMA address, the `handle_id` and `key` (the latter two are returned by `fork_prepare`) at the coordinator. We also record the time when the seed was deployed, which is necessary to prevent the coordinator forking from a near-expired cache instance. The seed store can be co-located with the coordinator or implemented as a distributed key-value store.

**Reclamation.** Similar to Caching, the long-lived seeds are reclaimed by timeout. Unlike Caching, seeds can have a much longer keep-alive time (e.g., 10 minutes vs. 1 minute) since they consume orders of magnitude smaller memory. The coordinators can renew the seed if it doesn’t live long enough for the forked function.

## 6.3 Fork tree and short-lived seed management

**Fork tree granularity and structure.** Each serverless workflow has a dedicated fork tree stored and maintained at the coordinator executing it. The upper-layer nodes in the tree correspond to the upstream functions (parents) in the workflow and the lower-layer nodes represent the downstream functions (children). Each node encodes the container IDs and locations, which is sufficient for the coordinator to reclaim the corresponding seed.

**Fork tree construction and destroy.** The construction of the fork tree is straightforward: After the coordinator forks a new child from a short-lived seed, it will add the seed to the tree. When all functions in the tree finish, MITOSIS will reclaim all the nodes except for the root node: the root node can be a long-lived seed and MITOSIS will not reclaim it.

**Fault tolerance.** The fork tree should be fault-tolerant to prevent memory leakage caused by dangling seeds. Replicating the tree with common replication protocols (e.g., Paxos [74]) can tolerate the failure, but adds non-trivial overheads during the workflow execution. Observing that serverless functions have a maximum lifetime (e.g., 15 minutes in AWS Lambda [3]), we use a simple timeout-based mechanism to tolerate the failures. Specifically, invokers will periodically garbage collect short-lived seeds if they run beyond the func-

tion’s maximum allowed runtime.

## 6.4 Limitation

First, fork still needs a long-lived seed to quickly bootstrap others. If no seed is available, we can leverage existing approaches that optimize coldstart (e.g., FaasNET [119]) to first start one. Second, fork only enables a read-only state transfer. Yet, it is sufficient for serverless workflow—the dominant function composition method. Finally, fork cannot transfer states between multiple upstream functions. Thus, MITOSIS must fuse these upstream functions into one or fallback to messaging (see `Portfolio` in Figure 3) for such cases. We are addressing this limitation by further introducing a *remote merge* primitive to complement the remote fork.

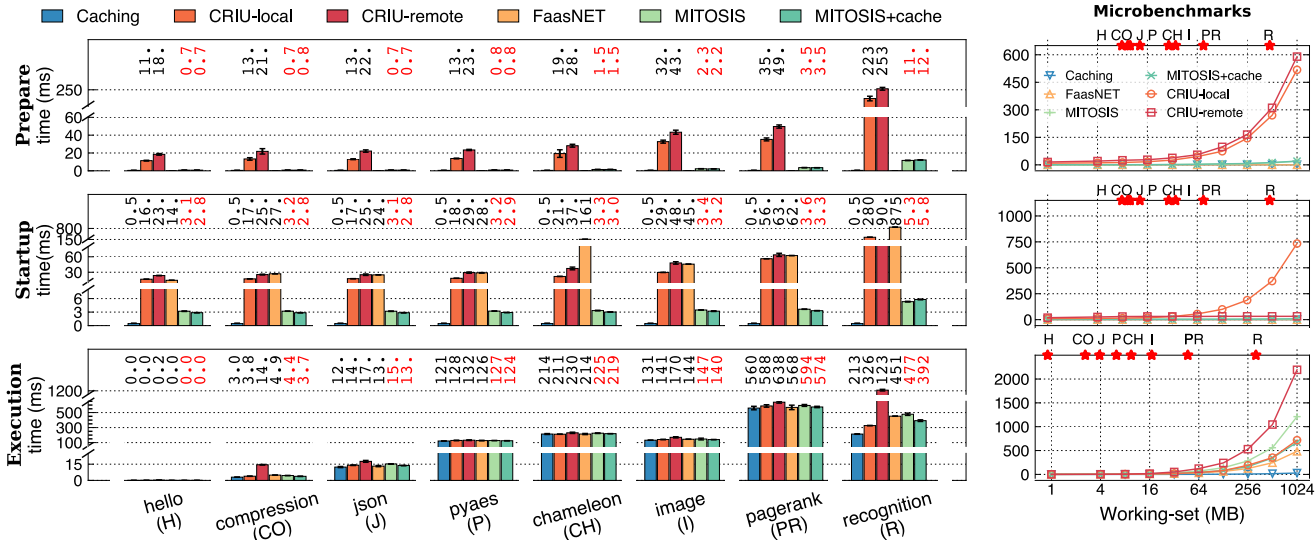
## 7 Evaluation

**Experimental setup.** We conduct all our experiments on a local cluster with 24 machines. Each machine has two 12-core Intel Xeon E5-2650 v4 processors and 128GB of DRAM. 16 machines are connected to two Mellanox SB7890 100Gbps switches with two 100 Gbps ConnectX-4 MCX455A Infini-Band RNICs. We use them as invokers to execute the serverless functions. Nodes without RDMA are left as coordinators.

**Comparing targets.** The evaluating setups of MITOSIS and its baselines are listed as follows. Note that we apply our generalized lean container (§5.2) to all the systems to hide the cost of containerization.

1. **Caching** is the de facto warmstart technique that provides a near-optimal function startup.
2. **CRIU-local** leverages CRIU [7] to implement remote fork (see Figure 5 (a)) and stores all files in an in-memory local filesystem (`tmpfs`). The file is transferred via our optimized transfer library with one-sided RDMA. We also apply existing on-demand restore optimization [120].
3. **CRIU-remote** leverages CRIU and a distributed file system for the remote fork (see Figure 5 (b)). We use Ceph [89]—a state-of-the-art production DFS that embraces RDMA. We also apply optimizations from CRIU-local: in-memory storage and on-demand restore.
4. **FaasNET** [119] optimizes the container image pulling of coldstart with function trees. We evaluate an optimal setup of FaasNET (for performance) that pre-provisions the images at all the invokers.<sup>8</sup>
5. **MITOSIS** is configured with on-demand execution and reads all pages from remote with a prefetch size of one.
6. **MITOSIS+cache** is the version of MITOSIS that always caches and shares the fetched pages among children. It essentially fallbacks to the local fork.

<sup>8</sup>The setup has been confirmed by the FaasNET authors.



**Figure 12.** (a) End-to-end latency comparisons of MITOSIS and baselines. (b) Analyses of different phases using microbenchmarks. Note that the working set of the execution is smaller than the prepare and startup because child only touches a subset of the parent’s memory.

**Functions evaluated.** We chose functions from representative serverless benchmarks (i.e., ServerlessBench [131], FunctionBench [68], and SeBS [31]), which cover a wide range of scenarios, including simple function (*hello/H*—print ‘Hello world’), file processing (*compression/CO*—compress a file), web requests (*json/J*—(de)serialize json data, *pyaes/P*—encrypt messages, *chameleon/CH*—generate HTML pages), image processing (*image/I*—apply image processing algorithms to an image), graph processing (*pagerank/PR*—execute the pagerank algorithm on a graph) and machine learning (*recognition/R*—image recognition using ResNet). These functions are written in python—the dominant serverless language [33]. Besides, we also use a synthetic *micro-function* that touches a variant portion of the memory to analyze the overhead introduced by MITOSIS. It is written in C to minimize the language runtime overhead interference.

### 7.1 End-to-end latency and memory consumption

Figure 12 shows the results of end-to-end latency: the left sub-figure is the time of different phases of the functions during remote fork, and the right is each phase’s result on micro-function. The function request is sent by a single client. To rule out the impact of disk accesses, we put all the function’s related files (e.g., images used by *image/I*) in tmpfs.

**Prepare time.** The prepare time is the time for the parent to prepare a remote fork. For CRIU-local and CRIU-remote, it is the time to checkpoint a container. For variants of MITOSIS, it is the `fork_prepare` time. Caching and FaasNET do not have this phase because they do not support fork.

MITOSIS is orders of magnitude faster in preparation than CRIU-local and CRIU-remote. On average, it reduces the prepare time by 94%. For example, MITOSIS prepared a 467 MB *recognition/R* container in 11 ms, while CRIU-local

and CRIU-remote took 223 ms and 253 ms, respectively. The variants of CRIU are bottlenecked by copying the container state from the memory to the filesystems.

**Startup time.** We measure the startup time as the time between an invoker receiving the function request and the time the first line of the function executes. As shown in the middle of Figure 12, caching is the fastest (0.5 ms) because starting a cached container only requires a simple unpauses operation. MITOSIS comes next, it can start all the functions within 6 ms. It is up to 99%, 94%, and 97% (from 98%, 86%, and 77%) faster than CRIU-local, CRIU-remote, and FaasNET, respectively. The startup time of MITOSIS is dominated by the generalized lean container setup time since reading the descriptor with RDMA is extremely fast with our fast descriptor fetch protocol.

The startup of CRIU-local is dominated by copying the entire file (shown in Figure 12 (b)). Using CRIU-remote avoids transferring the file, but the overhead of communicating with the DFS meta server (from 23–90 ms) is still non-trivial. Compared to CRIU-remote, MITOSIS can directly read the container metadata (descriptor) from the remote machine’s kernel. Finally, the startup cost of FaasNET (coldstart) is dominated by the runtime initialization of the function, as we skipped the image pull process of it. The overhead depends on the application characteristics. For example, *recognition/R* requires loading a ResNet model from PyTorch, which takes 875 ms. Other techniques can skip the loading process since the model has been loaded in the parents or the cached containers.

Note that the results of CRIU-remote and FaasNET are not significantly higher in the startup microbenchmark (Figure 12 (b)). For CRIU-remote, it is because the time (40ms) is relatively small compared to CRIU-local (>191ms for working-set larger than 256MB). For FaasNET, we use a native lan-

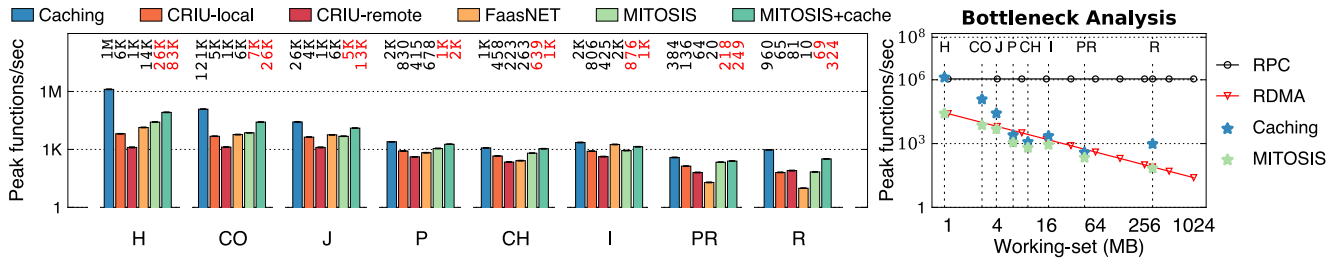


Figure 13. (a) Peak throughput comparisons of MITOSIS and baselines. (b) Bottleneck analysis of MITOSIS using a single parent seed.

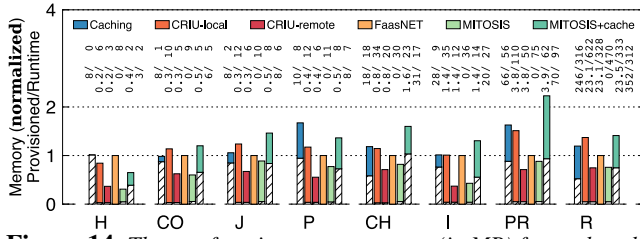


Figure 14. The per-function memory usage (in MB) for each technique before running (hatched) and during runtime (colored).

guage in the microbenchmark (C), so it doesn’t suffer from the runtime initialization and library loading costs of the application functions in Figure 12 (a).

**Execution time.** For function execution, MITOSIS is up to  $2.24\times$ ,  $1.46\times$  and  $1.14\times$  (from  $1.04\times$ ,  $1.04\times$ , and  $1.02\times$ ) slower than Caching, CRIU-local and FaasNET, respectively, except for *hello/H*. The overhead is mainly due to page faults and reading remote memory, which is proportional to the function working set (see Figure 12 (b)). Consequently, the overhead is most significant in *recognition/R* that reads 321 MB of the parent memory: MITOSIS is  $2.24\times$  (477 vs. 213 ms) and  $1.46\times$  (477 vs. 326 ms) slower than Caching and CRIU-local, respectively. CRIU-local is faster since it reads files from the local memory (tmpfs). To remedy this, MITOSIS+cache reduces the number of remote memory accesses by reading from the local cached copies of the remote pages. It improves performance by up to 17%, making MITOSIS close to or better than CRIU-local and FaasNET during execution. Note that Caching is always optimal (i.e., faster than FaasNET and CRIU-local) because it has no page fault overhead. Finally, MITOSIS is up to  $3.02\times$  (from  $1.02\times$ ) faster than CRIU-remote thanks to bypassing DFS for reading remote pages.

**Memory consumption.** Figure 14 reports the amortized per-machine memory consumed for each function categorized by provisioned memory (before running) and runtime memory. An ideal serverless platform should use minimal provisioned memory for each function. On average, MITOSIS only consumes 6.5% of the provisioned memory (one cached instance across 16 machines) while Caching requires at least 16 instances. CRIU-local/remote consumes a slightly lower memory (77% on average) than MITOSIS, because it reuses

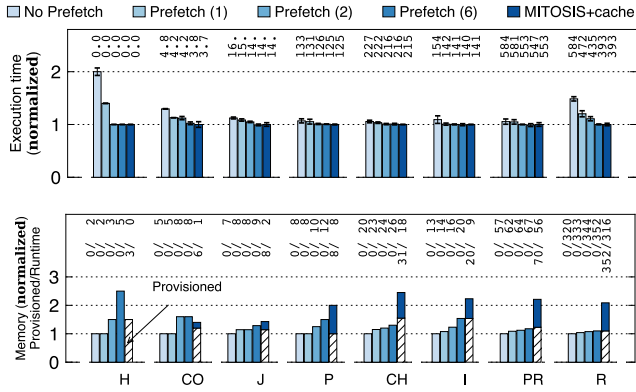
the local OS’s shared libraries to prevent storing them in the checkpointed files. This works at the cost of requiring storing all the function’s required libraries on all the machines, otherwise the restored container will fail. For the same reason, MITOSIS consumes a slightly larger runtime memory (8% on average) than CRIU-remote. Yet, its runtime memory is smaller than CRIU-local because the CRIU-local will read the entire file before it can execute the function.

## 7.2 Bottleneck analysis and throughput comparisons

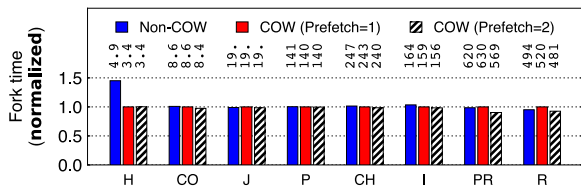
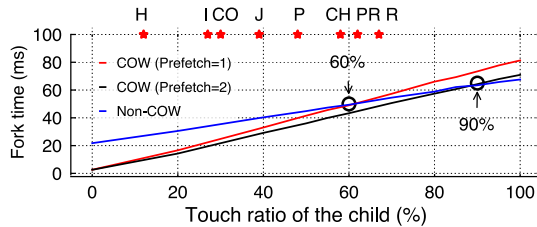
**Bottleneck analysis.** Using a single seed function is ideal for resource usage. However, the parent-side network bandwidth (RDMA) and two RPC threads can become the bottleneck. Meanwhile, MITOSIS is also bottlenecked by the aggregated client-side CPU resources processing the function logic. The peak client-side performance for each function is the peak throughput of running functions with Caching.

Figure 13 (b) analyzes the impact of the above factors. We utilize all 16 invokers to achieve the peak throughput. For H, CO, J, and R, RDMA is the bottleneck. For example, *recognition/R* touches 321 MB of the parent’s memory, so the RDMA (200 Gbps) can only serve (ideal) 80 forks/sec. Thus, MITOSIS achieves 69 reqs/sec and is lower than Caching (960 reqs/sec). In contrast, if the children CPU is the bottleneck, MITOSIS is similar to Caching (P, CH, I, and PR). For example, Caching can only execute 384 reqs/sec for *pagerank/PR*. In comparison, RDMA can handle an ideal 544 PR forks/sec (the working set is 47 MB). Thus, MITOSIS can achieve a slightly lower throughput (249 reqs/sec). Finally, the RPC would never become the bottleneck: two kernel threads can handle up to 1.1 million reqs/sec, which is always faster than RDMA for working set from 1 MB to 1 GB.

**Throughput comparison.** Figure 13 (a) further compares the peak throughput of different approaches. Note that we exclude the prepare phase of CRIU—otherwise, it will be bottlenecked by this phase. MITOSIS is up to  $8.0\times$  (from  $2.1\times$ ) faster than CRIU-local, thanks to avoiding the whole file during the restore phase. Compared with CRIU-remote, MITOSIS is also up to  $20.4\times$  (from  $2.1\times$ ) faster except for R (69 vs. 81): CRIU-remote reads a smaller amount of remote memory because it reuses local copies of the shared libraries. R has the largest working set, so it is mostly affected by the network. For the others, MITOSIS is faster as it bypasses the overhead of DFS.



**Figure 15.** Effects of the number of pages prefetched per-fault on (a) execution time (in ms) and (b) memory consumption (in MB).



**Figure 16.** Effects of COW to latencies on (a) the micro-function (with a 64 MB parent working set) and (b) serverless functions.

We omit the comparison between MITOSIS and Caching, which has been studied in the bottleneck analysis.

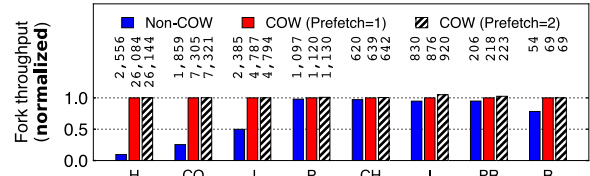
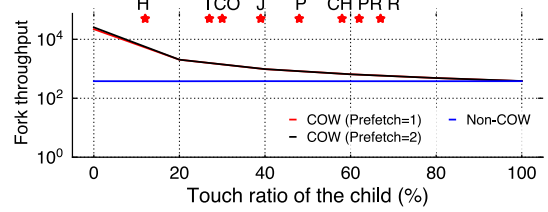
### 7.3 Effects of prefetching

We next explore how the prefetch number affects MITOSIS in Figure 15 (a). As we can see, prefetching can significantly improve the execution time of functions: prefetching 1, 2, and 6 pages improve the average time by 10%, 16%, and 18% (up to 30%, 50%, and 50%), respectively. More importantly, a small prefetch size (6) can achieve a near-identical performance as the optimal, i.e., no remote access, (MITOSIS+cache). Note that for small prefetch size the cost to the throughput is negligible, so we omit the results.

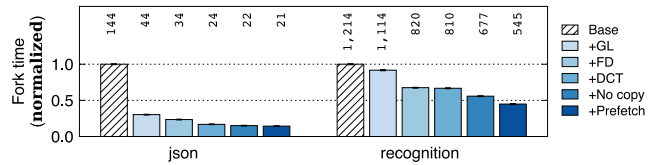
Prefetching has additional runtime memory consumption: as shown in Figure 15 (b), prefetching 1, 2, and 6 consumes average  $1.1\times$ ,  $1.3\times$ , and  $1.5\times$  (up to  $1.15\times$ ,  $1.6\times$ , and  $2.5\times$ ) more memory than no prefetching. Therefore, we currently adopt a prefetch size of 1 to reduce runtime memory usage.

### 7.4 Effects of copy-on-write (COW)

MITOSIS reads the child’s pages in an on-demand way (copy-on-write). This section presents the benefits and costs of COW



**Figure 17.** Effects of COW to peak thpt on (a) the micro-function (with a 64 MB parent working set) and (b) serverless functions.



**Figure 18.** Effects of optimizations applied by MITOSIS.

compared to a non-COW design—the child will read all the parent’s memory before executing the functions.

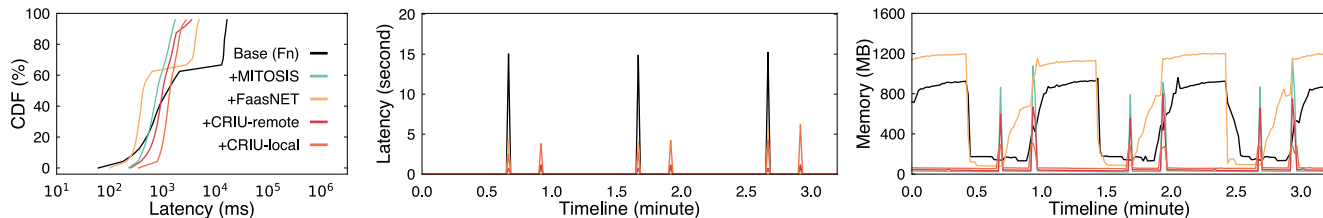
**Latency.** Figure 16 reports the latency results. The benefit of COW in latency depends on the amount of the parent’s memory touched by the child (touch ratio): the cross points in the microbenchmark are 60% and 90% when the prefetch size is 1 and 2, respectively. For larger prefetch size, the cross point is close to 100%. Non-COW has a longer startup time due to extra remote memory reading, but it is more efficient in reading pages with RDMA because it can batch multiple paging requests [66]. Nevertheless, serverless functions typically have a moderate touch ratio (i.e.,  $< 67\%$ ). Therefore, COW has averages of 8.7% (from 0.6% to 44%) and 3.7% (from -5% to 31%) lower latency than Non-COW when the prefetch size is 1 and 2, respectively.

**Throughput.** Figure 17 further reports the throughput results. Unlike latency, COW is always faster in throughput (except for 100% touch ratio) because non-COW will issue more RDMA requests. Consequently, COW is  $1.03\times$ – $10.2\times$  faster than Non-COW on serverless functions.

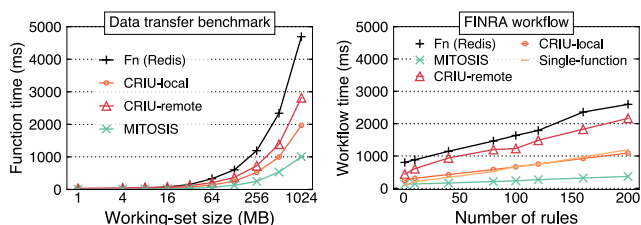
### 7.5 Effects of optimizations

Due to space limitation, Figure 18 briefly shows the effects of optimizations introduced in §5 on the end-to-end fork time using a short function (*json*/J) and a long function (*recognition*/R). First, generalized lean container (+GL) reduced a fixed offset of the latency (100 ms) to all the functions compared with a baseline of using runC [13]. Compared with RPC, fast descriptor fetch with one-sided RDMA (+FD) further contributes 10% and 25% latency reduction for both





**Figure 19.** (a) The latency CDF, (b) average latency, and (c) memory consumption timelines on image processing (I) under spikes.



**Figure 20.** (a) The state-transfer performance between two functions and (b) performance of FINRA.

functions. The improvement is more obvious for R because its descriptor is much larger (1.3 MB vs. 31 KB). Using DCT instead of RC reduced a 10–20 ms to the functions, and directly exposing the physical memory with RDMA instead of copying them (+no copy) further reduced the fork time by 12% and 20% for J and R, respectively. Finally, prefetching (+prefetch) shortens the time by 9% and 15%.

## 7.6 State-transfer performance

**Microbenchmark.** We use the data-transfer testcase (5) in ServerlessBench [131] to compare different approaches to transfer states between two remote functions. As shown in Figure 20 (a), MITOSIS is up to 1.4–5× faster than Fn, which leverages Redis to transfer data between functions, when transferring 1 MB–1 GB data. Note that we exclude the data (de)serialization overhead (by skipping the phase) and coldstart overhead (by pre-warming the containers) in Fn. Otherwise, the gap between Fn and MITOSIS would become larger. Compared to CRIU-local/remote, MITOSIS is faster thanks to the design for a fast remote fork (see §7.1).

**Application: FINRA.** We next present the performance of MITOSIS on FINRA [14], whose workflow graph is shown in Figure 2. We manually fuse the `fetchPortfolioData` and `fetchMarketData` into one function to fully leverage remote fork for MITOSIS and CRIU variants. For FN, functions use Redis to transfer states. Figure 20 (b) reports the end-to-end latency w.r.t the number of instances of `runAuditRule`, where FINRA spawns about 200 instances [10]. We select the market data from seven stocks, resulting in a total 6 MB states transferred between functions.

As we can see, MITOSIS is 84–86%, 47–66% and 71–83% faster than the baseline Fn, CRIU-local and CRIU-remote, respectively. Note that we have pre-warmed Fn to prevent the effects of coldstart—which is unnecessary for MITOSIS. Fn is bottlenecked by Redis (27 ms) and data serialization and

de-serialization (600 ms). MITOSIS has no such overhead and it further makes state transfer between machines optimal via RDMA. Moreover, MITOSIS can scale to a distributed setting with little COST [88]—it can outperform a single-function sequentially processing all the rules (Single-function). This is because MITOSIS can concurrently run functions across machines with minimal cost transferring data between them.

## 7.7 Performance under load spikes

Finally, we evaluate the performance of MITOSIS under load spike using *imageI* on the real-world traces (660323 [102]). Figure 19 (a) summarizes the latency CDFs. The 99<sup>th</sup> percentile latency of FN+MITOSIS is 73.64% and 89.08% smaller than FN+FaasNET and FN, respectively, thanks to avoiding the coldstart with remote fork. Nevertheless, its median latency is 1.85× longer than FaasNET (799 ms vs. 430 ms), because FaasNET leverages Caching and has a 65.1% cache hit during spikes. However, Caching incurs non-trivial memory consumption: Fn (and Fn+FaasNET) will cache a container for 30 seconds if it is a coldstart, resulting in a significant amount of memory usage (see Figure 19 (c)). In comparison, MITOSIS only caches a single seed and saves orders of magnitude memory during the idle time. For example, at time 2.3 min, MITOSIS only consumes 29 MB memory per-machine, which is 3% and 2% of Fn (914 MB) and Fn+FaasNET (1,199 MB), respectively.

## 8 Discussion

**Seed placement and selection policies.** We currently choose a random placement policy. A better policy may further consider network topology and system-wide load balance. Meanwhile, we simply choose the first container experiencing coldstart as the long-lived seed, yet, a better selection policy should further consider the status of the running container. For instance, recent works have discovered that containers may need multiple invocations to warm up properly [28, 107], e.g., to JIT a function written in a managed language. Therefore, choosing a properly warm-up container as the seed can significantly improve the function performance after the fork. As these policies are orthogonal to MITOSIS, we plan to investigate them in the future.

**Frequency and cost of fallbacks.** The frequency of fallbacks can significantly impact the performance of remote forks. During our experiment, we encountered no fallbacks

because the parent (cached container) had loaded all the children’s memory. However, fallbacks can happen in corner cases (e.g., swapping). The per-page overhead is  $22\times$  (65 vs.  $3\mu\text{s}$ ) due to the cost of RPC and loading the page from disk (SSD). Currently, one fallback handler can process 16 K paging requests per second, so it will not become a bottleneck.

**The benefits of implementing MITOSIS in the kernel.** We choose to implement MITOSIS in the kernel for performance considerations. First, a user-space solution cannot directly access the physical memory of the container, so it pays the checkpointing overhead (see §3). Moreover, the kernel can establish RDMA connections more efficiently (see KR-Core [11]), and the kernel-space page fault handler is much faster than the user-space fault handler.

## 9 Related Work

**Optimizing serverless computing.** MITOSIS continues the line of research on optimizing serverless computing, including but not limited to accelerating function startups [94, 17, 106, 37, 117, 101, 119], state transfer [110, 69, 96, 71, 17, 86], stateful serverless functions [132, 63], transactions [84], improving the cost-efficiency [134, 42, 100, 76, 98, 40, 38], and others [109, 133, 65, 36, 64, 15, 114, 85, 130, 136]. While most of these works are orthogonal to MITOSIS, we believe they can also benefit from our work. In particular, we propose using the remote fork abstraction to simultaneously accelerate function startups and state transfer, which is critical to all serverless applications. We also compare our work extensively to its closest related approaches in §2. Moreover, while the implementation of Linux fork may not be optimal in some scenarios [129, 24, 135], it has been shown to be suitable for serverless functions [17, 37]. Thus, we generalize the fork abstraction to accelerate functions running across machines.

**Checkpoint and restore (C/R).** C/R has been investigated by OSes for a long time [39, 82]. e.g., KeyKOS [51], EROS [104], Aurora [116] and others [52, 72, 7, 137, 118, 21, 26, 48]. Aurora [116] leverages C/R to realizing efficient single level store, it introduces techniques including system shadowing for efficient incremental checkpointing. MITOSIS eliminates checkpointing in the context of remote fork via OS-RDMA co-design. VAS-CRIU [118] also noticed the overhead of C/R introduced by filesystems. It leverages multiple independent address spaces (MVAS) [50] to bypass the filesystem for C/R on a single machine. We further use kernel-space RDMA to build a global distributed address space and scale fast C/R to a distributed setting.

**Remote fork (migrations).** Besides using C/R for remote fork [108, 32], MITOSIS is also inspired by works on virtual machine fork (SnowFlock [73]) and migrations [18, 29, 45, 56, 55, 92, 81], just to name a few. For example, the MITOSIS container descriptor is inspired by the VM descriptor used in SnowFlock, which only captures the critical metadata used for instantiating a child container at the remote side. We further

consider the opportunities and challenges when embracing RDMA for remote fork in the context of serverless computing. We believe our techniques can benefit existing works not utilizing RDMA.

**RDMA-based remote paging and RDMA multicast.** Reading pages from remote hosts via RDMA is not a so new technique in modern OSes [19, 46, 16, 87, 103]. For example, Infiniswap [46] leverages RDMA to build a fast swap device for memory disaggregation. Remote regions [16] proposes a remote file-like abstraction to simplify exposing an application’s memory with RDMA. MITOSIS further builds efficient remote fork by reading remote pages in a “copy-on-write” fashion with RDMA.

MITOSIS exhibits a pull-based RDMA multicast communication pattern, where multiple children pull from the same parent’s memory during load spikes. Push-based RDMA multicast has been extensively studied in the literature [25, 61, 62]. For example, RDMC [25] proposes a binomial pipeline protocol where a sender can efficiently push data to a group of nodes using RDMA. We believe MITOSIS can further benefit from research on pull-based RDMA multicast.

## 10 Conclusion

We present MITOSIS, a new OS primitive designed for fast remote fork by co-designing with RDMA. MITOSIS possesses two key attributes for serverless computing. (1) Startup efficiency: MITOSIS is orders of magnitude faster than coldstart while consuming orders of magnitude fewer resources than warmstart (with comparable performance). (2) State transfer efficiency: functions can directly access pre-materialized states from the forked function. Extensive evaluation using real-world serverless applications confirmed the efficacy and efficiency of MITOSIS on commodity RDMA-capable clusters. While we focus on serverless computing in this paper, we believe MITOSIS also shines with other tasks, e.g., container migrations.

## Acknowledgment

We sincerely thank our shepherd Christopher Rossbach and the anonymous reviewers, whose reviews and suggestions greatly strengthened our work. We also thank Wentai Li, Qingyuan Liu, Zhiyuan Dong, Dong Du, Nian Liu, Sijie Shen, and Xiating Xie for their valuable feedback. This work was partly supported by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 62202291, 62202292, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 22511106200), as well as research grants from Huawei Technologies and Shanghai AI Laboratory. Corresponding author: Rong Chen ([rongchen@sjtu.edu.cn](mailto:rongchen@sjtu.edu.cn)).

## References

- [1] Dynamically connected transport. [https://www.openfabrics.org/images/2018workshop/presentations/303\\_ARosenbaum\\_DynamicallyConnectedTransport.pdf](https://www.openfabrics.org/images/2018workshop/presentations/303_ARosenbaum_DynamicallyConnectedTransport.pdf), 2018.
- [2] Apache OpenWhisk Composer. <https://github.com/apache/openwhisk-composer>, 2022.
- [3] AWS Lambda FAQs. <https://aws.amazon.com/en/lambda/faqs/>, 2022.
- [4] AWS Step Functions. <https://aws.amazon.com/step-functions/>, 2022.
- [5] Ceph - a scalable distributed storage system. <https://github.com/ceph/ceph/tree/luminous-release>, 2022.
- [6] CRIU Lazy migration. [https://criu.org/Lazy\\_migration](https://criu.org/Lazy_migration), 2022.
- [7] CRIU Website. [https://www.criu.org/Main\\_Page](https://www.criu.org/Main_Page), 2022.
- [8] docker container pause. [https://docs.docker.com/engine/reference/commandline/container\\_pause/](https://docs.docker.com/engine/reference/commandline/container_pause/), 2022.
- [9] Docker Website. <https://www.docker.com/>, 2022.
- [10] FINRA adopts AWS to perform 500 billion validation checks daily. <https://aws.amazon.com/solutions/case-studies/finra-data-validation/>, 2022.
- [11] KRCORE: a microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association.
- [12] Provisioned concurrency for lambda functions. <https://aws.amazon.com/cn/blogs/aws/new-provisioned-concurrency-for-lambda-functions/>, 2022.
- [13] runc. <https://github.com/opencontainers/runc>, 2022.
- [14] United States Financial Industry Regulatory Authority. <https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/>, 2022.
- [15] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020), R. Bhagwan and G. Porter, Eds., USENIX Association, pp. 419–434.
- [16] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., NOVAKOVIC, S., RAMANATHAN, A., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 775–787.
- [17] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 923–935.
- [18] AL-KISWANY, S., SUBHRAVETI, D., SARKAR, P., AND RIPEANU, M. Vmflock: virtual machine co-migration for the cloud. In *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011* (2011), A. B. Maccabe and D. Thain, Eds., ACM, pp. 159–170.
- [19] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 14:1–14:16.
- [20] AO, L., PORTER, G., AND VOELKER, G. M. Faasnap: Faas made fast using snapshot-based vms. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 730–746.
- [21] ARMENATZOGLOU, N., BASU, S., BHANOORI, N., CAI, M., CHAINANI, N., CHINTA, K., GOVINDARAJU, V., GREEN, T. J., GUPTA, M., HILLIG, S., HOTINGER, E., LESHINKSY, Y., LIANG, J., MCCREEDY, M., NAGEL, F., PANDIS, I., PARCHAS, P., PATHAK, R., POLYCHRONIOU, O., RAHMAN, F., SAXENA, G., SOUNDARARAJAN, G., SUBRAMANIAN, S., AND TERRY, D. Amazon redshift re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022* (2022), Z. Ives, A. Bonifati, and A. E. Abbadi, Eds., ACM, pp. 2205–2217.
- [22] AWS. Aws fargate. <https://aws.amazon.com/cn/fargate/>, 2022.
- [23] AWS. Aws lambda. <https://aws.amazon.com/lambda>, 2022.
- [24] BAUMANN, A., APPAVOO, J., KRIEGER, O., AND ROSCOE, T. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 14–22.
- [25] BEHRENS, J., JHA, S., BIRMAN, K., AND TREMEL, E. RDMC: A reliable RDMA multicast for large objects. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018* (2018), IEEE Computer Society, pp. 71–82.
- [26] BILAL, M., CANINI, M., FONSECA, R., AND RODRIGUES, R. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. *CoRR abs/2105.14845* (2021).



- [27] CADDEN, J., UNGER, T., AWAD, Y., DONG, H., KRIEGER, O., AND APPAVOO, J. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 32:1–32:15.
- [28] CARREIRA, J., KOHLI, S., BRUNO, R., AND FONSECA, P. From warm to hot starts: leveraging runtimes for the serverless era. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021* (2021), S. Angel, B. Kasikci, and E. Kohler, Eds., ACM, pp. 58–64.
- [29] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings* (2005), A. Vahdat and D. Wetherall, Eds., USENIX.
- [30] CLOUD, A. Alibaba serverless application engine. <https://www.aliyun.com/product/aliware/sae>, 2022.
- [31] COPIK, M., KWASNIEWSKI, G., BESTA, M., PODSTAWSKI, M., AND HOEFLER, T. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021* (2021), K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds., ACM, pp. 64–78.
- [32] CRIU. CRIU Usage scenarios. [https://criu.org/Usage\\_scenarios](https://criu.org/Usage_scenarios), 2022.
- [33] DATADOG. The state of serverless. <https://www.datadoghq.com/state-of-serverless/>, 2022.
- [34] DOCKER. Docker Registry. <https://docs.docker.com/registry/>, 2022.
- [35] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.
- [36] DU, D., LIU, Q., JIANG, X., XIA, Y., ZANG, B., AND CHEN, H. Serverless computing on heterogeneous computers. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 797–813.
- [37] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.
- [38] DUKIC, V., BRUNO, R., SINGLA, A., AND ALONSO, G. Photons: lambdas on a diet. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 45–59.
- [39] EGWUTUOHA, I. P., LEVY, D., SELIC, B., AND CHEN, S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* 65, 3 (2013), 1302–1326.
- [40] FINGLER, H., AKSHINTALA, A., AND ROSSBACH, C. J. USETL: unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2019, Hangzhou, China, Augsut 19-20, 2019* (2019), ACM, pp. 23–30.
- [41] FOR AWS LAMBDA CONTAINER REUSE, B. P. <https://medium.com/capital-one-tech/best-practices-for-aws-lambda-container-reuse-6ec45c74b67e>, 2022.
- [42] FUERST, A., AND SHARMA, P. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. Berger, and C. Kozyrakis, Eds., ACM, pp. 386–400.
- [43] GAO, Y., LI, Q., TANG, L., XI, Y., ZHANG, P., PENG, W., LI, B., WU, Y., LIU, S., YAN, L., FENG, F., ZHUANG, Y., LIU, F., LIU, P., LIU, X., WU, Z., WU, J., CAO, Z., TIAN, C., WU, J., ZHU, J., WANG, H., CAI, D., AND WU, J. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 519–533.
- [44] GOOGLE. Google serverless computing. <https://cloud.google.com/serverless>, 2022.
- [45] GU, J., HUA, Z., XIA, Y., CHEN, H., ZANG, B., GUAN, H., AND LI, J. Secure live migration of SGX enclaves on untrusted cloud. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017* (2017), IEEE Computer Society, pp. 225–236.
- [46] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), A. Akella and J. Howell, Eds., USENIX Association, pp. 649–667.
- [47] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.
- [48] GUO, Z., BLANCO, Z., SHAHRAD, M., WEI, Z., DONG, B., LI, J., POTTA, I., XU, H., AND ZHANG, Y. Resource-centric serverless computing, 2022.



- [49] GUO, Z., SHAN, Y., LUO, X., HUANG, Y., AND ZHANG, Y. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 417–433.
- [50] HAJJ, I. E., MERRITT, A., ZELLWEGER, G., MILOJICIC, D. S., ACHERMANN, R., FARABOSCHI, P., HWU, W. W., ROSCOE, T., AND SCHWAN, K. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016* (2016), T. Conte and Y. Zhou, Eds., ACM, pp. 353–368.
- [51] HARDY, N. Keykos architecture. *SIGOPS Oper. Syst. Rev.* 19, 4 (oct 1985), 8–25.
- [52] HARGROVE, P. H., AND DUELL, J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series* (2006), vol. 46, IOP Publishing, p. 067.
- [53] HELLERSTEIN, J. M., FALEIRO, J. M., GONZALEZ, J., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019), www.cidrdb.org.
- [54] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016* (2016), A. Clements and T. Condie, Eds., USENIX Association.
- [55] HINES, M. R., DESHPANDE, U., AND GOPALAN, K. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.* 43, 3 (jul 2009), 14–26.
- [56] HINES, M. R., AND GOPALAN, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009* (2009), A. L. Hosking, D. F. Bacon, and O. Krieger, Eds., ACM, pp. 51–60.
- [57] HONG, Y., ZHENG, Y., YANG, F., ZANG, B., GUAN, H., AND CHEN, H. Scaling out numa-aware applications with rdma-based distributed shared memory. *J. Comput. Sci. Technol.* 34, 1 (2019), 94–112.
- [58] HUAWEI. Huawei cloud functions. <https://developer.huawei.com/consumer/en/agconnect/cloud-function/>, 2022.
- [59] IBM. IBM Cloud Functions. <https://www.ibm.com/cloud/functions>, 2022.
- [60] INTEL. Intel 64 and IA-32 Architectures Software Developer’s Manual. <https://cdrdv2.intel.com/v1/dl/getContent/671200>, 2022.
- [61] JHA, S., BEHRENS, J., GKOUNTOUVAS, T., MILANO, M., SONG, W., TREMEL, E., VAN RENESSE, R., ZINK, S., AND BIRMAN, K. P. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.* 36, 2 (2019), 4:1–4:49.
- [62] JHA, S., ROSA, L., AND BIRMAN, K. Spindle: Techniques for optimizing atomic multicast on RDMA. *CoRR abs/2110.00886* (2021).
- [63] JIA, Z., AND WITCHEL, E. Boki: Stateful serverless computing with shared logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 691–707.
- [64] JIA, Z., AND WITCHEL, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 152–166.
- [65] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019* (2019), ACM, pp. 158–164.
- [66] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.
- [67] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.
- [68] KIM, J., AND LEE, K. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019* (2019), ACM, p. 477.
- [69] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. *login Usenix Mag.* 44, 1 (2019).
- [70] KNATIVE. <https://knative.dev>, 2022.
- [71] KOTNI, S., NAYAK, A., GANAPATHY, V., AND BASU, A. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 805–820.
- [72] LAADAN, O., AND HALLYN, S. E. Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium* (2010), vol. 159, Citeseer.

- [73] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009* (2009), W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, Eds., ACM, pp. 1–12.
- [74] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (2001), 51–58.
- [75] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.* 7, 4 (1989), 321–359.
- [76] LI, Q., LI, B., MERCATI, P., ILLIKKAL, R., TAI, C., KISHINEVSKY, M., AND KOZYRAKIS, C. RAMBO: resource allocation for microservices using bayesian optimization. *IEEE Comput. Archit. Lett.* 20, 1 (2021), 46–49.
- [77] LINUX. Kernel samepage merging. <https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html>, 2022.
- [78] LINUX. Linux swap. <https://www.linux.com/news/all-about-linux-swap-space/>, 2022.
- [79] LINUX. Page migration. [https://www.kernel.org/doc/html/latest/vm/page\\_migration.html](https://www.kernel.org/doc/html/latest/vm/page_migration.html), 2022.
- [80] LINUX. Transparent hugepage. <https://www.kernel.org/doc/html/latest/vm/transhuge.html>, 2022.
- [81] LITZKOW, M., AND SOLOMON, M. Supporting checkpointing and process migration outside the unix kernel.
- [82] LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. Checkpoint and migration of unix processes in the condor distributed processing system. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1997.
- [83] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 773–785.
- [84] LYKHENKO, T., SOARES, R., AND RODRIGUES, L. Faastcc: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference* (New York, NY, USA, 2021), Middleware '21, Association for Computing Machinery, p. 159–171.
- [85] LYU, X., CHERKASOVA, L., AITKEN, R. C., PARMER, G., AND WOOD, T. Towards efficient processing of latency-sensitive serverless dags at the edge. In *EdgeSys@EuroSys 2022: Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking, Rennes, France, April 5-8, 2022* (2022), A. Y. Ding and V. Hilt, Eds., ACM, pp. 49–54.
- [86] MAHGOUN, A., SHANKAR, K., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 285–301.
- [87] MARUF, H. A., AND CHOWDHURY, M. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 843–857.
- [88] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015* (2015), G. Candea, Ed., USENIX Association.
- [89] MELLANOX. Bring up ceph rdma - developer's guide. <https://community.mellanox.com/s/article/bring-up-ceph-rdma---developer-s-guide>, 2021.
- [90] MELLANOX. Kernel verbs api update. <https://www.openfabrics.org/images/eventpresos/2016presentations/204KernelVerbs.pdf>, 2022.
- [91] MICROSOFT. Azure functions. <https://azure.microsoft.com/en-us/services/functions/>, 2022.
- [92] MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration. *ACM Comput. Surv.* 32, 3 (2000), 241–299.
- [93] NVIDIA. Rdma aware networks programming user manual. <https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=34256548>, 2022.
- [94] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTE, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.
- [95] PONS, D. B., ARTIGAS, M. S., PARÍS, G., SUTRA, P., AND LÓPEZ, P. G. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019* (2019), ACM, pp. 41–54.
- [96] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 193–206.
- [97] QI, S., MONIS, L., ZENG, Z., WANG, I., AND RAMAKRISHNAN, K. K. SPRIGHT: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022* (2022), F. Kuijpers and A. Orda, Eds., ACM, pp. 780–794.

- [98] QIU, H., BANERJEE, S. S., JHA, S., KALBARCZYK, Z. T., AND IYER, R. K. FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 805–825.
- [99] ROTHENBERGER, B., TARANOV, K., PERRIG, A., AND HOEFLER, T. ReDMark: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 4277–4292.
- [100] RZADCA, K., FINDEISEN, P., SWIDERSKI, J., ZYCH, P., BRONIEK, P., KUSMIEREK, J., NOWAK, P., STRACK, B., WITUSOWSKI, P., HAND, S., AND WILKES, J. Autopilot: workload autoscaling at google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 16:1–16:16.
- [101] SAXENA, D., JI, T., SINGHVI, A., KHALID, J., AND AKELLA, A. Memory deduplication for serverless computing with medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 714–729.
- [102] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.
- [103] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. Le-goos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 69–87.
- [104] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999* (1999), D. Kotz and J. Wilkes, Eds., ACM, pp. 170–185.
- [105] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 317–332.
- [106] SHILLAKER, S., AND PIETZUCH, P. *FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing*. USENIX Association, USA, 2020.
- [107] SHIN, W., KIM, W., AND MIN, C. Fireworks: a fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 663–677.
- [108] SMITH, J. M., AND IOANNIDIS, J. *Implementing remote fork () with checkpoint/restart*. Department of Computer Science, Columbia Univ., 1987.
- [109] SREEKANTI, V., WU, C., CHHATRAPATI, S., GONZALEZ, J. E., HELLERSTEIN, J. M., AND FALEIRO, J. M. A fault-tolerance shim for serverless computing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 15:1–15:15.
- [110] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIER-SMITH, J., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452.
- [111] TARANOV, K., ROTHENBERGER, B., PERRIG, A., AND HOEFLER, T. Srdma: Efficient nic-based authentication and encryption for remote direct memory access. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USA, 2020)*, USENIX ATC'20, USENIX Association.
- [112] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 199–212.
- [113] THOMAS, S., AO, L., VOELKER, G. M., AND PORTER, G. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 16–29.
- [114] THORPE, J., QIAO, Y., EYOLFSON, J., TENG, S., HU, G., JIA, Z., WEI, J., VORA, K., NETRAVALI, R., KIM, M., AND XU, G. H. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (2021), A. D. Brown and J. R. Lorch, Eds., USENIX Association, pp. 495–514.
- [115] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.
- [116] TSALAPATIS, E., HANCOCK, R., BARNES, T., AND MASH-TIZADEH, A. J. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 788–803.
- [117] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 559–572.

- [118] VENKATESH, R. S., SMEJKAL, T., MILOJICIC, D. S., AND GAVRILOVSKA, A. Fast in-memory CRIU for docker containers. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019* (2019), ACM, pp. 53–65.
- [119] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 443–457.
- [120] WANG, K. A., HO, R., AND WU, P. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 39:1–39:16.
- [121] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 133–146.
- [122] WEBSITE, A. O. <https://openwhisk.apache.org>, 2022.
- [123] WEBSITE, F. P. <https://fnproject.io>, 2021.
- [124] WEI, X., CHEN, R., AND CHEN, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.
- [125] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 233–251.
- [126] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.
- [127] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving Graph Traversal with Split Live Migration. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), pp. 723–738.
- [128] XING, J., HSU, K.-F., QIU, Y., YANG, Z., LIU, H., AND CHEN, A. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 2585–2600.
- [129] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2313–2328.
- [130] YANG, Y., ZHAO, L., LI, Y., ZHANG, H., LI, J., ZHAO, M., CHEN, X., AND LI, K. Influss: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 768–781.
- [131] YU, T., LIU, Q., DU, D., XIA, Y., ZANG, B., LU, Z., YANG, P., QIN, C., AND CHEN, H. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 30–44.
- [132] ZHANG, H., CARDOZA, A., CHEN, P. B., ANGEL, S., AND LIU, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1187–1204.
- [133] ZHANG, W., FANG, V., PANDA, A., AND SHENKER, S. Kappa: a programming framework for serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 328–343.
- [134] ZHANG, Y., GOIRI, I. N., CHAUDHRY, G. I., FONSECA, R., ELNIKETY, S., DELIMITROU, C., AND BIANCHINI, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 724–739.
- [135] ZHAO, K., GONG, S., AND FONSECA, P. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 540–555.
- [136] ZHAO, L., YANG, Y., LI, Y., ZHOU, X., AND LI, K. Understanding, predicting and scheduling serverless workloads under partial interference. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021* (2021), B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, pp. 22:1–22:15.
- [137] ZHONG, H., AND NIEH, J. Crak: Linux checkpoint/restart as a kernel module. Tech. rep., Citeseer, 2001.





# Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)

Baptiste Lepers  
*University of Neuchâtel*

Willy Zwaenepoel  
*University of Sydney*

## Abstract

We demonstrate that hardware management of a tiered memory system offers better performance for many applications than current methods of software management. Hardware management treats the fast tier as a cache on the slower tier. The advantages are that caching can be done at cache line granularity and that data appears in fast memory as soon as it is accessed. The potential for cache conflicts has, however, led previous works to conclude these hardware methods generally perform poorly.

In this paper we show that low-overhead conflict avoidance techniques eliminate conflicts almost entirely and thereby address the above limitation. We explore two techniques. The static technique tries to avoid conflicts between pages at page allocation time. The dynamic technique relies on monitoring memory accesses to distinguish between hot and cold pages. It uses this information to avoid conflicts between hot pages, both at page allocation time and by dynamic remapping at runtime.

We have implemented these techniques in the Linux kernel on an Intel Optane machine in a system called Johnny Cache (JC). We use HPC applications, key-value stores and databases to compare JC to the default Linux tiered memory management implementation and to HeMem, a state-of-the-art software management approach.

Our measurements show that JC outperforms Linux and HeMem for most applications, in some cases by up to 5×. A surprising conclusion of this paper is that a cache can provide close-to-optimal performance by minimizing conflicts purely at page allocation time, without any access monitoring or dynamic page remapping.

## 1 Introduction

Tiered memory systems combine DRAM with a slower, but more abundant, storage tier (SSD, PMEM, CXL memory extension modules [8], ...). Most systems rely on a software daemon that monitors accesses to the data. Frequently accessed

data is migrated to DRAM, while less frequently accessed data is migrated to the slower tier [1, 9, 11, 14, 20, 23, 25]. Tiered memory systems have also been implemented purely in hardware, using DRAM as an "L4" cache that sits between the CPU and the slower tier [13].

Previous work has argued that hardware implementations of tiered systems are inefficient because the hardware lacks a high-level view of the application requirements and because caching strategies have to be kept simple to be executed in hardware. For instance, in tiered DRAM+PMEM systems, software daemons have been shown to outperform the "memory mode" of Intel CPUs (in "memory mode", the CPU uses DRAM as a directly-mapped cache for PMEM) [20].

This paper is based on the observation that the previously mentioned limitations of hardware caching are not fundamental and can be addressed at the operating system level. In particular, we demonstrate that the poor performance observed in earlier hardware-based systems is due to cache conflicts resulting from Linux's page allocation policy, and that simple improvements to the page allocation policy can reduce cache conflicts with little or no overhead.

Linux's page allocation does not take into consideration the location of pages in hardware caches. As a consequence, Linux suffers from the birthday paradox: the DRAM cache is large, but many pages tend to map to a subset of the available cache locations. We propose the following simple static page allocation policy to reduce conflicts: we allocate a new page such that its physical address maps to a cache slot with the fewest pages currently mapped to it. For example, if we have a cache with 2 million slots and 4 million pages to be allocated, we allocate 2 pages to each slot. The static policy has no noticeable overhead, but it vastly reduces conflicts.

We also investigate a dynamic policy that takes into account the access frequency of pages, distinguishing between hot and cold pages. The dynamic policy allocates a new page to the cache slot with the lowest access frequency and reacts to workload changes by dynamically remapping pages when it detects conflicts. Surprisingly, we find that in many workloads the static policy already results in few conflicts, and the

overheads of the dynamic policy offset the benefits of any further gains in conflict reduction.

We compare our conflict avoidance policies to software migration, as proposed by, among others, HeMem [20]. With software migration, access frequency is monitored as well, producing the same set of hot and cold pages and incurring the same monitoring overhead as our dynamic approach. Software migration, however, uses this information for an entirely different purpose, namely to migrate hot pages from slow to fast memory, and vice versa for cold pages, unlike our dynamic policy which uses it to reduce conflicts.

We have implemented the static and dynamic policies at the kernel level in a subsystem named Johnny Cache (JC), and we refer to these systems as JC-static and JC-dyn, respectively. We have evaluated these systems on a tiered DRAM+PMEM system against the Linux page allocation mechanism and against HeMem, a state-of-the-art software-based page migration system [20]. JC outperforms Linux and HeMem for the vast majority of applications, in some cases by up to  $5\times$ . We document these results in more detail in the paper and also discuss the limitations of a cache-based approach. In addition, we find that JC-static often suffices to obtain good performance. Methods involving profiling such as HeMem and JC-dyn suffer from profiling and migration overheads and the inability to detect hot pages in some workloads. In contrast, avoiding conflicts in the DRAM cache at allocation time, as done by JC-static, is robust and sufficient to achieve near-optimal performance for most workloads.

In summary, the paper makes the following contributions:

- The observation that hardware-managed DRAM caches can be made efficient by minor modifications to the operating system page allocation algorithms.
- The idea of placing conflict avoidance as a first principle of page management in tiered memory systems, instead of relying on migration of data.
- The design, implementation and evaluation of page placement policies that outperform state-of-the-art page migration systems.

The rest of the paper is organized as follows. Section 2 explains how tiered-main memory systems are managed in software and in hardware. Section 3 presents the design of our policies, Section 4 presents their implementation, and Section 5 their evaluation. Section 6 provides further discussion of the strengths and weaknesses of various approaches. Section 7 presents related work and Section 8 concludes.

## 2 Tiered main memory systems

In this section, we give an overview of existing software- and hardware-managed tiered memory systems, and we compare their overheads.

### 2.1 Software-based migration

In software-managed tiered memory systems, the operating system chooses which pages are allocated in DRAM and which pages are allocated in the slower tier. The kernel usually allocates as many pages in DRAM as possible and, when DRAM is full, subsequent pages are allocated in the slow tier. A daemon is in charge of migrating frequently accessed pages (hot pages) from the slow tier to DRAM, and infrequently accessed pages (cold pages) from DRAM to the slow tier. The techniques vary but aim at inferring the set of hot pages with high accuracy and low overhead. For instance, HeMem [20] uses the hardware performance unit of Intel CPUs to track memory accesses and migrates pages between DRAM and PMEM using DMA to minimize CPU overheads.

Software-based migration gives the operating system full control over page placement, but it comes with some downsides. First, data migrations are costly because they can only happen at page granularity (4KB or 2MB), and each migration requires modifying the page table, modifying the kernel VMA metadata and flushing the TLBs. Migrations may also cause latency spikes in write-heavy applications because pages have to be write-protected while being migrated. Second, since access frequency is collected on a per-page basis, for applications that mix hot and cold data in the same page, DRAM may need to be used for cold data to allow fast access to hot data in the same page. Finally, page migrations happen asynchronously: data may be accessed for a while in the slow tier before being migrated to DRAM. As a consequence, the performance of software-based migration is heavily dependent on the fast and accurate detection and migration of the working set. To do so, memory access must be sampled with high frequency, a costly proposition.

### 2.2 Hardware caching

In hardware, the CPU uses DRAM as a cache for the slow tier. In existing implementations [13], the DRAM is configured as a 1-way cache, indexed by physical address. Unlike software-based approaches, hardware caches are synchronous: all accessed data is cached in DRAM. In this section, we describe the implementation of the "memory mode" of Intel processors for tiered DRAM+PMEM systems.

When looking for a physical address  $W$ , the memory controller first checks if  $W$  is in the DRAM cache (at location " $W \bmod \text{cachesize}$ "). If  $W$  is not present in DRAM,  $W$  is fetched from PMEM, copied to the DRAM cache and to the CPU cache (see Figure 1(a)).

A conflict occurs when  $W$  maps to a cache slot that is already occupied by  $X$ , in which case  $X$  must first be removed from the cache. In the best case,  $X$  is clean, and the cost is equal to that of a PMEM read. If  $X$  is dirty, then  $X$  must be written back to PMEM before  $W$  can be loaded in the cache, making the cost the sum of a PMEM write and a PMEM

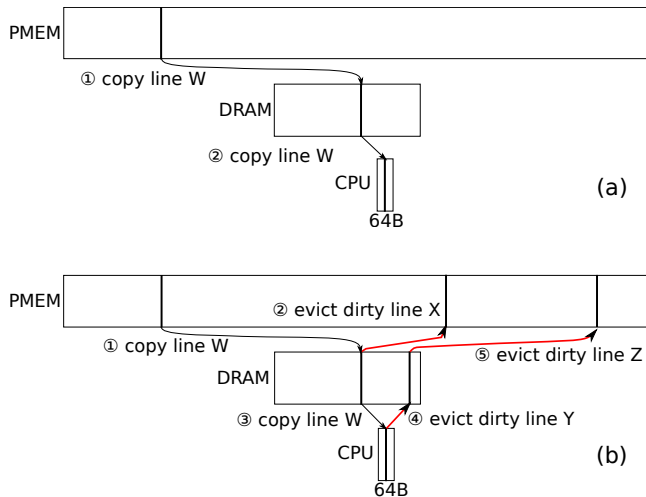


Figure 1: Caching in memory mode. (a) In the best case data found in PMEM is cached in DRAM and in the CPU caches, and in the (b) worst case the caching may result in evictions, causing up to two writebacks to PMEM.

read. A worst-case scenario can arise as described in Figure 1(b). In addition to the writeback of *X*, a dirty line *Y* may be evicted from the CPU cache, resulting in a writeback of *Y* to the DRAM cache, which itself may result in a writeback of another dirty line *Z* to PMEM (the DRAM cache is not inclusive).

Regardless of the precise sequence of events, conflicts are expensive. Table 1 compares the latency of performing 8B random reads or random writes in DRAM, in PMEM, and in the worst-case scenario presented in Figure 1(b). Reading from PMEM (in AppDirect mode) is  $3.2\times$  slower than reading from DRAM, and writing is  $4.4\times$  slower. A read causing two writebacks to PMEM is  $9.7\times$  slower than a read from DRAM, and a write causing two writebacks is  $7.2\times$  slower than a write to DRAM. (Causing two writebacks to PMEM is not exactly equivalent to performing two writes to PMEM, which would be  $8.8\times$  slower, because the CPU overlaps the evictions with other processing done by the application, resulting in slightly more in-CPU parallelism).

Memory mode thus performs suboptimally when frequently accessed data conflicts in the cache. If, however, conflicts can be avoided, then memory mode offers several advantages over software migration. First, caching avoids costly whole-page migration as well as virtual memory operations. Second, caches operate at the cache line level, while software migration can only migrate data at the page granularity. Therefore, they avoid wasting DRAM space if hot and cold data are located in the same page. Finally, caching is synchronous: hot data appears in DRAM on the first access.

Read in DRAM	96ns
Plain read from PMEM	305ns
Write in DRAM	130ns
Plain write from PMEM	578ns
Read/Write causing 2 writebacks	938ns

Table 1: Latency of memory access in various scenarios.

## 2.3 Comparison

Table 2 summarizes the costs of migrating data vs. caching data. These costs show that caching data in DRAM is *a priori* a more parsimonious solution: data is cached at cache line granularity (vs. page granularity), caching data requires no kernel metadata updates, and no memory profiling is necessary to infer which pages to cache.

## 3 Design

Our design is based on the idea that a DRAM cache is efficient, as long as conflicts in the cache are rare. Conflicts happen when two data items are mapped to the same cache location. Conflicts become problematic if the data items are accessed in turn. We have designed two policies that aim at minimizing conflicts in the cache. The hardware caches data at cache line granularity, but the kernel can only allocate data at page granularity, so our policies try to minimize conflicts between pages.

**Static policy:** The static policy minimizes the number of allocated pages that map to the same DRAM cache location. Assuming a DRAM cache that can store  $D$  pages, the static policy allocates the first  $D$  pages so that they map to different cache locations. The next  $D$  pages are allocated so that they possibly conflict with a single other page, and so on.

**Dynamic policy:** The dynamic policy samples memory accesses to compute the heat of every page and every cache location. When a new page is allocated, the kernel maps it to the coldest available location.

A conflict avoidance daemon monitors for conflicts between hot pages at the same cache location. When two pages, mapped to the same DRAM cache location, are both frequently accessed, one of the pages is remapped to a different cache location.

**Rationale:** The main advantage of the static policy is that it requires no monitoring of memory accesses, and so it runs with no overhead. The intuition behind the static policy is that minimizing the number of pages that overlap in the cache reduces significantly the likelihood of conflicts between hot data items. Indeed, in most workloads, at most of a few GBs of data is hot, even in workloads whose memory footprint vastly exceeds the available DRAM. Minimizing overlaps makes conflicts between hot pages unlikely. For instance, let's consider an application that allocates data twice the available DRAM size, 5% of which is hot. The static policy allocates



	Software migration	Hardware caching
Granularity	Page Size (4KB, 2MB)	64B
Cost of migrating/caching	Page copy from PMEM to DRAM Page copy from DRAM to PMEM 2 page table updates 2 VMA updates TLB flush	<i>Best case</i> 1 cache line copied from PMEM to DRAM <i>Worst case</i> 1 cache line copied from PMEM to DRAM 2 cache lines copied from DRAM to PMEM
Strategy	Software defined	All memory accesses are cached in DRAM

Table 2: Comparison of the cost of migration vs. caching.

pages such that exactly 2 pages map to each cache location. A given hot page has a 5% chance to compete for the cache with another hot page, and a 95% chance to compete with a cold page. In other words, most hot pages are "paired" with a cold page, and are thus unlikely to be evicted frequently from the DRAM cache.

The dynamic policy makes more informed choices at page allocation time, and the daemon fixes conflicts that may have been missed at allocation time. The dynamic policy borrows the notion of heat from software migration systems, but the heat is used to track and *avoid conflicts* between hot pages rather than to migrate hot pages to DRAM. We demonstrate in Section 5.2 that, in the general case, avoiding conflicts is less costly than migrating hot data to DRAM.

## 4 Implementation

In this section, we describe the implementation of the page allocation policies and the migration daemon. The code is available at <https://github.com/BLepers/JohnnyCache>.

### 4.1 Page initialization and associated metadata

Our policies are implemented in the kernel, as hooks in the kernel initialization function, the page initialization function, the page fault handler and the page unmap handler. To ease the development of policies, we implemented a framework that contains the logic common to the policies. In the remainder of the paper, we refer to the framework as Johnny Cache (*JC*).

In this paper, we assume a directly-mapped 1-way cache, in which data is cached at its physical address modulo the size of the cache – as implemented by Intel in the "memory mode" of tiered DRAM+PMEM systems. It would be easy to account for associativity in *JC* by changing the definition of a conflict: currently, a conflict involves 2 or more pages; in an *N*-way cache, a conflict would involve *N*+1 or more pages.

While DRAM caches data at cache line granularity, the kernel can only allocate and migrate data at page granularity. All the metadata maintained by *JC* are thus at the page level. When the kernel boots, we query the processor's memory controller to find out the size of the DRAM cache. In the remainder of this section, we refer to the maximum number of

pages that the cache can hold as the cache *capacity*. Because the cache is directly-mapped, every page in the system maps to a unique index in the cache, which we call a *bin*. The bin of a page is its page frame number (physical address of the first byte of the page / size of a page) modulo the capacity of the cache. Furthermore, each bin of the cache has a *heat*. The definition of heat depends on the policy. For instance, for the static policy it corresponds to the number of allocated pages that map to that bin.

As is the case with the default page allocation policy of Linux, we use a lazy page allocation mechanism: pages are physically allocated only when they are first accessed. We thus hook the page fault handler to implement our page placement policies. The framework maintains a list of bins with available pages, sorted per heat. When a page fault occurs, a page from a bin with the lowest heat is returned, and the current allocation policy is informed of the page fault. Similarly, whenever a page is freed, the kernel unmap handler is called, and the current allocation policy is made aware of the unmapping.

Listing 1 summarizes the metadata and code of the page fault hook used by our framework. The overhead of keeping the metadata in memory is small (less than 50MB for a system with 128GB of DRAM and 1TB of PMEM).

Our policies are implemented at the kernel level and oblivious to the notion of a thread or an application. The policies try to minimize conflicts across the entire machine, and no partitioning of the cache is done (unlike page-coloring approaches). A major benefit of this approach is that conflicts are minimized globally. For instance, the conflict avoidance daemon remaps hot conflicting pages even if they belong to different applications.

### 4.2 Static policy

The static policy allocates a new page in a bin with the fewest allocated pages. The static policy consists of counting the number of allocated pages that map to a given cache bin. The policy is called on every page fault and page unmap by the framework. Listing 2 summarizes the code of the static policy. During a page fault, the policy increments by one the *heat* of the bin of the newly allocated page. Because the page fault handler of the common framework allocates a page from the bins which have the lowest heat, subsequent page faults will

---

**Listing 1** JC framework.

---

```
1 // struct for each cache bin (page granularity)
2 struct bin* bins[CACHE_CAPACITY] = { ... };
3 // avail[heat] = list of bins with free pages
4 struct bin* avail_bins[HEAT_LEVELS];
5 // full[heat] = list of fully allocated bins
6 struct bin* full_bins[HEAT_LEVELS];
7
8 struct page* page_fault_handler(void) {
9     for(i = 0; i < HEAT_LEVELS; i++) {
10        bin = list_first(avail_bins[i]);
11        if(bin) {
12            struct page *p =
13                list_pop(bin->avail_pages);
14            current_policy.page_fault(bin, p);
15            if(list_empty(bin->avail_pages))
16                list_move(bin, full_bins[bin->heat]);
17            else
18                list_move(bin, avail_bins[bin->heat]);
19            return p;
20        }
21    }
22    // OOM
23 }
24
25 void page_unmap(struct page *p) { ... }
```

---

likely avoid that bin. When a page is unmapped, the heat of the bin is decremented by one, increasing the likelihood of that bin being chosen for subsequent allocations.

---

**Listing 2** In the static policy, the heat of a bin corresponds to the number of pages allocated to that bin.

---

```
1 void static_pf(struct bin *b, struct page *p) {
2     b->heat++;
3 }
4
5 static_policy = {
6     .page_fault = static_pf, .unmap = ...
7 };
```

---

### 4.3 Dynamic policy and migration daemon

The dynamic policy allocates a page in a bin with available pages and with the lowest heat. The dynamic policy monitors memory accesses to infer the heat of each page and bin. We monitor read accesses to the DRAM cache, read accesses to PMEM and all stores using the Processor Event-Based Sampling (PEBS) feature of Intel's CPUs. When a memory access is sampled, we increase the heat of the accessed page and accessed bin. We also artificially increase the heat of the bin of newly allocated pages to avoid multiple pages being mapped to the same bin during bursts of allocations. To avoid heat continually increasing over time, we trigger page cooling as soon as a page becomes "super hot", i.e., when its heat becomes double that of the threshold to detect a hot page.

In theory, our dynamic policy could monitor conflicts in the cache instead of monitoring memory accesses, but no such event exists in Intel CPUs. Our heat detection and cooling approaches are identical to those used by HeMem [20], which allows for a fair comparison between software-based migration and conflict-avoidance (both solutions use the same PEBS events, the same definition of heat and the same cooling function).

The migration daemon monitors conflicts between allocated pages. When two hot pages are present in the same bin, one of them is remapped to a physical location in a different bin. The daemon periodically looks for pages in the upper heat buckets and remaps them. The remapping operation calls the page fault handler which allocates a new page in a cold bin and calls the unmap function, which decreases the heat of the original bin. Algorithm 3 summarizes the approach of the dynamic policy and migration daemon.

HeMem triggers migrations as soon as a hot page is detected. To allow for a fair comparison, we trigger the daemon as soon as we detect a bin containing two hot pages (i.e., as soon as we detect a conflict that involves two hot pages).

---

**Listing 3** The dynamic policy and migration daemon that remaps pages from highly accessed bins.

---

```
1 // Migration daemon
2 void migration_daemon() {
3     wait();
4     for(i=HEAT_LEVELS-1; i>MIN_CONTENTION; i--) {
5         foreach(bin, avail_bins[i]) {
6             if(bin->nb_hot_pages >= 2)
7                 remap(get_hot_page(bin));
8         }
9         foreach(bin, full_bins[i]) {
10            if(bin->nb_hot_pages >= 2)
11                remap(get_hot_page(bin));
12        }
13    }
14 }
15
16 // Called on every sampled memory access
17 void add_sample(struct bin *b, struct page *p){
18     b->heat++;
19     ... // increase the page's heat &
20     ... // update the metadata
21     if(b->nb_hot_pages > 2)
22         migration_daemon.wakeup();
23 }
24
25 void dyn_pf(struct bin *b, struct page *p) {
26     b->heat++;
27     ... // increase the page's heat &
28     ... // update the metadata
29 }
30
31 dynamic_policy = {
32     .page_fault = dyn_pf,
33     .unmap = ...
34 };
```

---

## 5 Evaluation

The evaluation aims at answering the following questions:

- What is the performance of JC compared to state-of-the-art tiered memory management systems?
- What is the overhead of JC, in terms of performance and latency spikes, compared to other systems?
- What are the limitations of JC? Which applications benefit from hardware caches, and which benefit from page migration?

We show that the static page allocation policy of JC achieves close-to-optimal performance in many applications, and sometimes outperforms the dynamic policy (and related work) when minimizing CPU overhead is essential for performance. The surprising conclusion of this evaluation is thus that hardware caches often outperform existing software-based migration strategies, provided minimal changes in the kernel page allocation policy are put in place.

### 5.1 Setup

**Hardware configuration.** All the experiments presented in this paper are run on a two-node NUMA machine, with 40 Intel Xeon Gold 6230 cores running at 2.10GHz (20 cores per NUMA node), 128GB of DRAM, and 8\*128GB Intel Optane NV-DIMMs (64GB DRAM and 512GB PMEM per NUMA node). In memory mode, each NUMA node has a DRAM cache of 48GB (16GB is used by the CPU for the cache metadata).

**Workloads.** We borrow the workloads used to evaluate HeMem [20], a state-of-the-art software page migration system. The GUPS microbenchmark allocates a large array, zeroes it, and then threads perform updates to a random subset of 8-byte array elements. BC, from the GAP benchmark suite, computes the betweenness centrality algorithm on a powerlaw graph [4]. Silo [22] is an in-memory database running the standard TPC-C benchmark suite. Finally, Masstree [16] is an in-memory key-value store, running a YCSB workload<sup>1</sup>. We also present results from the NAS benchmark suite [3]. JC equals or more commonly outperforms the related work in all these benchmarks, with the exception of the MG.E application from the NAS benchmark suite, which allows us to demonstrate the limitations of our approach. As in the original HeMem evaluation, NUMA effects of tiered memory are beyond the scope of this paper, and we run the applications on a single NUMA node.

**Software configuration.** We compare JC against unmodified Linux and HeMem. *Linux* uses the machine in memory mode

<sup>1</sup>HeMem benchmarked FlexKVS [17], an in-memory key-value store which we could not evaluate due to the lack of an RDMA network card on our server.

with the default Linux page allocation policy. With the default page allocation policy, pages are allocated on the local NUMA node, but contiguous virtual memory ranges may end up fragmented in physical memory. Any array larger than 2 pages may thus conflict with itself in the DRAM cache (the larger the array, the more likely 2 pages of the array conflict). We refer to *JC-static* as the machine in memory mode with our static page allocation policy and to *JC-dyn* as the machine in memory mode with the dynamic page allocation policy plus the page remapping daemon. We benchmark HeMem using the provided artifact [19].

### 5.2 GUPS

The GUPS microbenchmark, from HeMem [20], allocates a large array, a subset of which is hot. 90% of the updates are done on the hot section of the array, and 10% on the cold section. We configure the array to be 96GB, twice the DRAM cache size and measure performance when 10% of the array is hot (9.6GB).

The performance of HeMem and JC-dyn is dependent on their ability to detect hot and cold pages. Intuitively, the more threads perform memory accesses, the easier it is to detect hot pages. To assess the impact of the workload on the detection of hot and cold pages, we thus vary the number of threads. Furthermore, HeMem and JC-dyn use two separate threads to sample memory accesses and to migrate pages. To assess the overhead of these threads, we either run them on separate cores or on the cores used by GUPS. We refer to these configurations as (M+N) where M is the number of cores used by GUPS, and N is either 0 or 2 and reflects the number of cores dedicated to monitoring and migration in HeMem and JC-dyn. We use three such configurations: (16+2), (8+2) and (8+0). These results are presented in Section 5.2.1.

In the original GUPS implementation the hot and cold data items are located in separate regions of the array. While this microbenchmark reflects the partitioning done by some applications, it implies that all hot items are located in a small number of pages, and all cold items are located in the other pages. To reflect the behavior of applications that do not partition their hot and cold items in this manner, we also run GUPS with hot items scattered randomly in the array. These results are presented in Section 5.2.2. Finally, in Section 5.2.3 we explore the performance of the different systems with a larger data set of 480GB.

We measure the throughput achieved for a given combination of system and workload. When the hot data fits in DRAM, we present the results as the percentage of the throughput achieved when all hot data is manually allocated in DRAM. Otherwise, we present the result in terms of millions of updates per second.

### 5.2.1 Random updates to clustered hot values

We first benchmark GUPS with hot values clustered on a few pages, the scenario favoring page migration.

The allocated array is twice the size of the DRAM cache so, initially, half of the array is in DRAM, and all systems start with low throughput. Figure 2 presents the performance of JC-static, JC-dyn, HeMem, and Linux over time, as a percentage of the performance achieved when all hot pages are manually allocated in DRAM.

**Steady-state performance in configuration (16+2):** Both HeMem and JC-dyn achieve 100% of DRAM performance, JC-static 85% and Linux 60%. There is (conflict-free) space in the cache for all hot pages, so JC-dyn eventually eliminates all conflicts, and HeMem eventually moves all hot pages to DRAM, explaining their performance being equal to DRAM.

The good performance of JC-static is explained by the low number of conflicts on hot data. The array is twice the size of the cache, and JC allocates exactly 2 pages per cache bin. Let  $P$  be a hot page.  $P$  conflicts with a single other page  $Q$ , and  $Q$  only has a 10% probability of being hot. JC thus only suffers from conflicts between 10% of the hot data (0.96GB).

The low performance of Linux is explained by the large number of conflicts when no care is taken to properly spread the pages in the cache. Just as with the "birthday paradox", even though a year has many days (*even though the cache has many bins*), in a small group of people, many are likely born on the same day (*many pages are mapped to the same cache bin, even when allocating only 100GB*). On average, Linux uses only 32GB of the DRAM cache because the allocated pages map to a subset of the available cache bins, while JC takes advantage of the full cache. Figure 2(d) presents the performance of an average run of Linux, but, depending on page placement, performance varies between runs from 20% to 80% of DRAM performance. These extreme values are rare, with most runs achieving around 60%.

**Steady-state performance in configuration (8+2) - The difficult configuration of heat detection systems:** The performance of JC-static and Linux relative to DRAM performance remains the same as in configuration (16+2). HeMem does not reach steady state even after 2 minutes, only reaching 40% of DRAM speed. JC-static is between  $4\times$  faster than HeMem (at the beginning of the execution) and  $2.2\times$  faster (after 2 minutes of execution). JC-dyn also does not reach steady state, but its performance is closer to DRAM performance (85% at the beginning of the execution, 90% at the end).

With 8 threads, fewer samples are generated, and the cooling mechanisms of HeMem and JC-dyn reduce the heat of pages faster than it increases as a result of accesses. HeMem and JC-dyn trigger page cooling as soon as any given page becomes "super hot", i.e., when its heat becomes double the threshold to detect a hot page (see Section 4). We implemented other cooling algorithms, but none ended up working

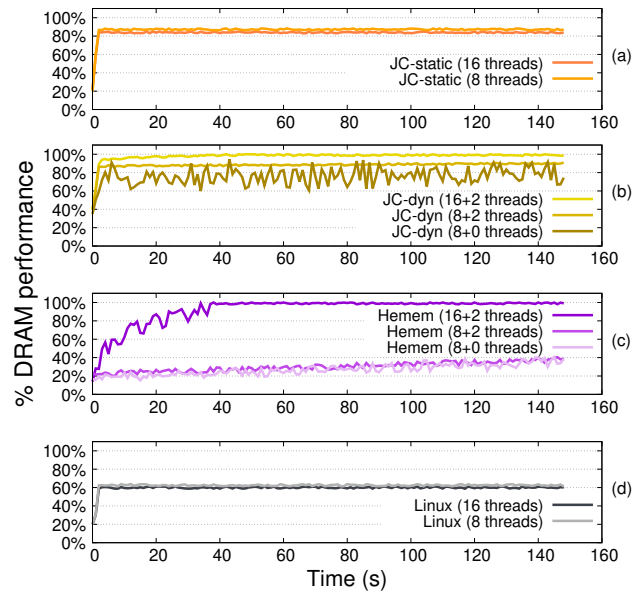


Figure 2: Hot values clustered in the array. Performance achieved by JC, HeMem, and Linux compared to the performance achieved when all the hot values are manually allocated in DRAM (optimal page placement). (a) JC-static, which does not use any profiling, performs close-to-optimally. (b) The profiling threads of the migration daemon can negatively interfere with the application threads (brown line). (c) The performance of HeMem is highly dependent on its ability to detect hot pages. When GUPS is launched with a low number of threads, hot pages are rarely detected and HeMem performs suboptimally. (d) The default page placement policy of Linux performs suboptimally because of conflicts.

in all configurations of GUPS. For instance, forcing cooling to happen periodically but less frequently results in most of the array being detected as hot in some other configurations (e.g., on smaller arrays). Replacing periodic cooling with other algorithms such as LRU also performs poorly in some configurations (e.g., when the hot set size exceeds the DRAM size). Most of the related work on page migration explores new ways of measuring heat accurately and with low overhead [1, 9, 11, 12, 14, 20, 23–25] but, in our experience, these heuristics require fine-tuning for each application and configuration.

It is possible to tune the sampling rate of memory accesses to gather more samples in a given amount of time, but doing so is also fraught with problems. For instance, doubling the sampling rate actually decreases the performance of GUPS running with 16 threads by 20%, due to profiling overheads. Some literature describes attempts to use dynamic sampling rates, but these algorithms also need to be precisely tuned for each machine or workload (e.g., to detect pages that need to be migrated between NUMA nodes, Carrefour [11] adjusts



its sampling rate based on the workload, but all its parameters are fine-tuned for each machine).

The inability to detect most hot pages negatively impacts the performance of HeMem. In comparison, caches "work well", even without any heat detection or page migration. Once an item is updated, it is cached in DRAM. After a few seconds, most items have been updated, and the cache has reached its warmed-up state. JC-static and JC-dyn perform close-to-optimally without any fine-tuning, regardless of the number of GUPS threads. Just as HeMem, JC-dyn only manages to detect a subset of the (conflicting) hot pages. The partial detection of hot pages explains why JC-dyn is unable to reach optimal performance, but also explains why it performs slightly better than JC-static.

**Performance over time in configuration (16+2) - Caches reach steady-state performance faster:** As expected, the performance of JC-static and Linux remains constant after allocation is completed, since the number of conflicts remains the same throughout the execution. HeMem and JC-dyn require some time to reach maximum performance, in the case of JC-dyn to migrate pages to avoid conflicts, in the case of HeMem to migrate hot pages initially allocated in PMEM to DRAM and vice versa for cold pages. The time to reach this steady state performance is, however, much longer for HeMem than for JC-dyn, 38 seconds versus 2 seconds.

HeMem needs to sample memory accesses in order to perform informed migration decisions, and each migration consists in evicting a cold page to PMEM and promoting a hot page to DRAM. Migrations are thus inherently asynchronous and costly. In JC, once an item is updated, it is cached in DRAM. After a few seconds, most items have been updated, and the cache has reached its warmed-up state.

A surprising observation is that fixing conflicts requires fewer page migrations than migrating hot and cold pages. Indeed, before doing any page migration, only 0.96GB of the data conflicts, and these conflicts can be avoided by migrating 0.48GB of data. In HeMem, 4.8GB of the hot data is misplaced and needs to be brought to DRAM, which also causes 4.8GB of cold data to be migrated to PMEM. HeMem thus migrates 9.6GB (20× more data) to reach steady state performance.

**Performance over time in configuration (8+0) - A background daemon can be counterproductive:** JC-static and Linux do not use any profiling threads, and their performance is obviously the same as in configuration (8+2). The performance of both HeMem and JC-dyn becomes quite variable, and JC-dyn on average drops below JC-static in terms of performance. When the monitoring and migration threads execute on dedicated cores, JC-static and JC-dyn have similar performance, but, when they are scheduled on the same cores as the GUPS application, they have a non-negligible impact on performance. In that situation, JC-static outperforms JC-dyn by 10% on average and maintains a much more stable

throughput over time.

## 5.2.2 Random updates to distributed hot values

In the previous experiment, all the hot items were clustered on the same pages, which is the best case scenario for page migration systems. However, hot items may not be clustered together in memory. To account for this behavior, we execute GUPS with hot items randomly scattered in the allocated array. As before, we allocate a 96GB array, 10% of which is hot. We execute GUPS with 16 threads, and dedicated cores for the profiling (16+2 configuration). Figure 3 presents the performance of HeMem, Linux and JC over time. JC is 4.5× faster than HeMem.

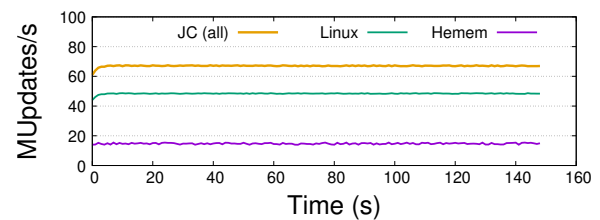


Figure 3: 16+2 threads, hot values distributed in the array and profiling running on dedicated cores. GUPS throughput over time when the hot set size is equal to 20% of the DRAM cache (higher is better).

In this experiment, because hot items are scattered in the array, most pages contain one or a few hot items. HeMem cannot bring all the hot pages in DRAM because the number of pages that contain hot items exceeds the number of pages that fit in DRAM. Interestingly, the hot data set does not need to be large for HeMem to be unable to migrate data to DRAM. Indeed, 1% of the data being hot (1GB) translates to 134 million 8-byte values, so all pages of the array likely contain many hot values (100GB is "only" 51K 2MB pages).

Just as HeMem, JC-dyn cannot perform any useful remappings, and as a result JC-static and JC-dyn perform equally well in this benchmark. Although hot items are scattered on all pages, since data is cached at the cache line granularity, the hot items rarely overlap in the cache. In this configuration of GUPS, JC reads on average 7× less data from PMEM than HeMem.

## 5.2.3 Performance on large datasets

In the previous experiments, GUPS was configured with a 96GB dataset (2× the cache size), 10% of which was hot. In this experiment, we configure GUPS to use 480GB (10× the cache size), the maximum workload size that fits on a single NUMA node, and we vary the percentage of hot data so that the hot data either fits in the cache or not. We run HeMem with dedicated cores for profiling and migration.

Because GUPS allocates all the available memory, JC's static page allocation policy and Linux have the same performance. Indeed, JC allocates pages in an order that minimizes conflicts but, in the end, both JC and Linux end up allocating all the pages of the system.

**When the hot dataset is clustered and fits in the cache.**

We measure performance when 2% of the data is hot (9.6GB, same as in the smaller experiments). Figure 4 presents the performance of JC and HeMem compared to the performance obtained when the hot data is placed in DRAM.

As in the smaller dataset experiment, HeMem and JC's dynamic performance depends on their ability to detect hot pages. When GUPS is configured to run with 8 threads, HeMem does not reach steady state after 2 minutes of execution. In comparison, caches "work well", even without heat detection or page migration.

When GUPS is configured with 16 threads, both HeMem and JC-dyn eventually reach optimal performance. As seen earlier, caches reach optimal performance faster because avoiding conflicts requires fewer page migrations (1.7GB of the data initially conflicts in JC, 4.3GB of the data is initially misplaced in HeMem). The number of conflicting pages is higher in the 480GB experiment than in the 96GB experiments, even though the same number of pages are hot, because more pages map to the same slot. In the 96GB experiment, 2 pages map on a given slot, so a hot page has a 10% probability of conflicting with another hot page. In the 480GB experiment, a hot page conflicts with 9 other pages, each of which has a 2% probability of being hot, so it has a 16% probability of conflicting with another hot page. The higher number of conflicts explains why JC-static performs worse on bigger datasets than on smaller datasets: large datasets hinder the ability of JC-static to minimize conflicts at allocation time.

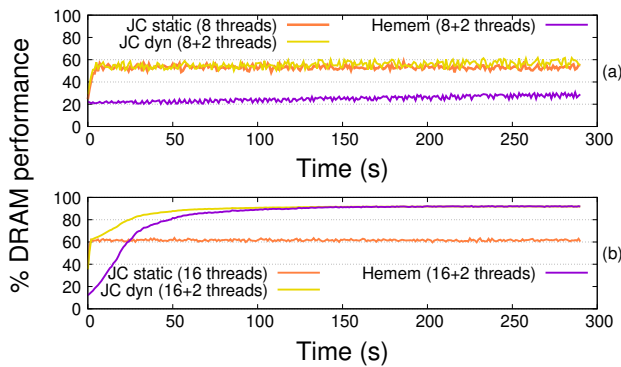


Figure 4: Hot values clustered in the array. 480GB dataset, 2% of which is hot (9.6GB). (a) With few threads, hot pages are rarely detected, and HeMem performs suboptimally. (b) With a large number of threads, JC reaches optimal performance faster than HeMem.

**When the hot dataset is clustered but does not fit in the cache.**

We measure performance when 50% of the data is hot (240GB, 5x the cache size). Figure 5 presents the performance of JC and HeMem compared to the performance that GUPS would get if all the data were to fit in DRAM. Interestingly, HeMem's performance slightly increases at the beginning of the benchmark as it brings hot pages in the DRAM cache. JC's performance slightly decreases as the cache fills with dirty data. Regardless of the policy, GUPS end up doing most of its memory accesses in PMEM because most of the data does not fit in the cache. In their steady state, all solutions have the same performance.

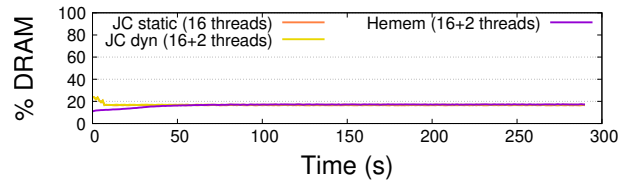


Figure 5: Hot values clustered in the array. 480GB dataset, 50% of which is hot (250GB). JC and HeMem do most of their accesses in PMEM because the hot dataset vastly exceeds DRAM capacity, which explains the low overall performance.

**When the hot dataset is not clustered.**

Figure 6 presents the performance of GUPS when the hot data is not clustered. As in the smaller experiment, HeMem cannot bring the hot data to DRAM and performs most of its accesses in PMEM. The performance of JC depends on the likelihood of conflicts. When a small percentage of the data is hot (2%, 9.6GB), then conflicts in the cache are unlikely. When most of the data is hot, JC also performs most of its accesses in PMEM and has the same performance as HeMem.

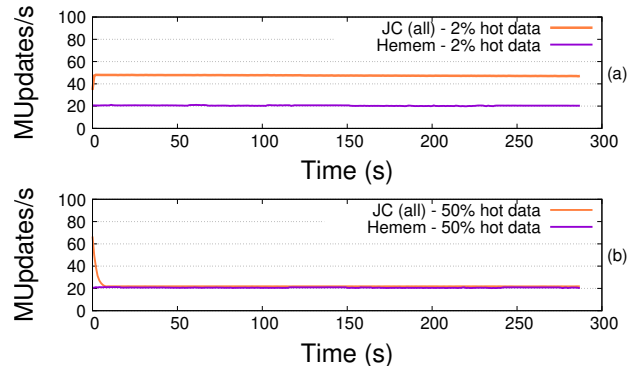


Figure 6: Hot values distributed in the array, 480GB dataset. The performance of JC depends on the percentage of hot values. Because hot values are spread on most pages, HeMem cannot improve performance and performs suboptimally.

## 5.2.4 Summary

In summary:

- + Hardware caches perform well, even without any active monitoring and page remapping. Software migration is highly dependent on its ability to detect hot pages.
- + Hardware caches reach steady-state performance faster than software migration.
- + Hardware caches often vastly outperform software migration when working with scattered small hot items.

## 5.3 BC

In this section, we evaluate the performance of BC, running with as many threads as cores, using the default Linux page allocator, HeMem and JC. Figure 7 presents the average duration of an iteration of the betweenness centrality algorithm.

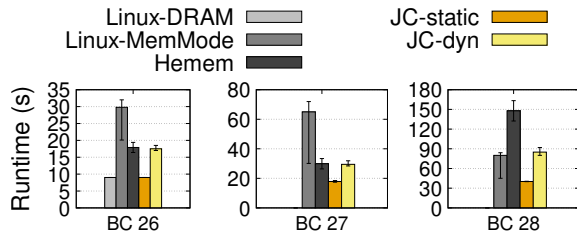


Figure 7: BC, average duration of an iteration, in seconds (lower is better). Linux-DRAM is the performance of BC when all data is manually allocated in DRAM. Linux-DRAM runs out of memory on BC 27 and BC 28. JC-static outperforms HeMem and JC-dyn.

**When the hot data fits in DRAM** With a scale of 26, BC allocates 35GB of memory and fits in DRAM. With a scale of 27, BC allocates 70GB, but then only actively accesses 45GB, so its "hot" dataset also fits in DRAM.

We confirm the results of the original HeMem paper: HeMem is faster than Linux in these two configurations. However, JC-static outperforms Linux and HeMem by up to  $3.2\times$  and  $2\times$  respectively. On BC 26, JC-static matches the performance of manually allocating all the data in DRAM (Linux-DRAM in Figure 7).

The performance differences are explained by the nature of the processing performed by BC. BC is an OpenMP application, and each of its threads performs a fixed fraction of the computation. The monitoring used by HeMem and JC-dyn uses two CPU-intensive threads, and these two threads compete for CPU with BC's threads. Because BC's threads frequently wait for each other in barriers, interrupting a single thread causes the whole application to be delayed at barriers. When run with HeMem or JC-dyn, BC 26 spends 50% of

its time waiting at barriers. In comparison, JC-static has no overhead during the execution of BC, BC's threads progress at the same pace and spend only 2.5% of their time at barriers.

The BC example again illustrates the difficulty of fine-tuning software-migration systems. In BC, we found that the optimal performance was reached when dividing the default sampling rate by  $10\times$  and performing page cooling once every second. In that configuration, the HeMem and JC-dyn versions of BC 26 match that of JC-static in performance, and for BC 27 they improve from 60% slower to 10% slower. However, with such a low sampling rate, no hot page is detected in the previously tested configurations of GUPS, resulting in JC-static being  $4\times$  faster than HeMem in that benchmark.

The poor performance of Linux is explained by conflicts in the DRAM cache. Conflicts between hot pages are rare (on average 500MB of hot pages conflict in BC 26), but these conflicts are not evenly distributed between threads: some threads end up manipulating pages that mostly conflict, while others manipulate pages that mostly do not conflict. Threads impacted by conflicts slow down the whole application because the fast threads spend most of their time waiting at barriers. We measured that threads spend on average 63% of their time waiting at barriers in BC 26.

**When the hot data does not fit in DRAM** With a scale of 28, BC allocates 140GB of memory and accesses 90GB of it. In this configuration, JC-static is  $5\times$  faster than HeMem, and HeMem is slower than the default Linux page allocation mechanism.

The low performance of HeMem and JC-dyn is again explained by the interference between their monitoring threads and BC, and pressure put by page migrations on PMEM. HeMem copies data from DRAM to PMEM using DMA, which has low CPU overhead, but still increases contention on PMEM. HeMem ends up migrating 40GB of data during the execution of BC 28, continuously putting pressure on PMEM, and exacerbating the imbalance issues observed at scale 26 and 27. At scale 28, on average threads spend 65% of their time waiting at barriers. JC-dyn performs better than HeMem on BC 28 because it creates less contention on PMEM: JC-dyn only infrequently migrates data (on average 4GB per run). Its overhead comes mostly from monitoring memory accesses and cooling pages.

JC-static performs well because it is able to avoid most conflicts at allocation time. Indeed, BC mostly operates on two arrays: a 10GB array is frequently accessed, the other one less so. JC-static allocates the pages of the frequently accessed array so that they do not conflict with each other, effectively minimizing conflicts without the need for any migration. It may seem "lucky" that the hot data was allocated at once, which causes JC-static to place all hot pages in different cache bins, but we found this pattern to be extremely common in HPC applications (e.g., all the NAS applications start by allocating large arrays, only a subset of which are hot).

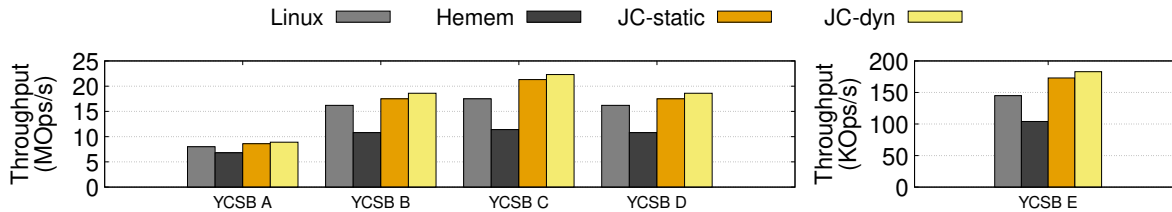


Figure 8: Masstree throughput (higher is better).

**Summary** Software-based migration requires monitoring memory accesses to perform informed migrations. The overhead of this monitoring can have cascading effects on HPC applications that rely on barriers to synchronize their threads. In contrast, it is possible to minimize conflicts in hardware caches at allocation time, without any profiling. Hardware caches thus vastly outperform software migrations when CPU overheads need to be avoided.

## 5.4 Masstree

We configured Masstree to execute with a 120 GB database, and we use 16 threads to avoid competition for CPU between Masstree’s threads and the profiling and migration threads, which run on dedicated cores. The YCSB workload used by Masstree uses 128B items (1 billion items in total) and follows a Zipfian distribution: 20% of the dataset is accessed 80% of the time. Most of the accesses thus target the index (5GB) and a subset of the values (25GB).

### 5.4.1 Performance

**Throughput** Figure 8 summarizes the performance of Masstree on the YCSB workload. In both applications, JC outperforms Linux and HeMem by up to  $2\times$ .

The memory access behavior of Masstree is similar to that of GUPS when the hot items are randomly scattered in the allocated array. During initialization, items are inserted in the key-value store in random order. It is thus possible for a hot value to be allocated next to a cold value. Similarly, the nodes of the index are populated in random order, and it is possible for a hot node to sit next to a cold one. Because the hot data is scattered on all pages, it is not possible to bring the hot dataset to DRAM.

In the case of GUPS with distributed hot values, HeMem could not improve the performance of the application at all because hot items were uniformly hot. In Masstree, the index is slightly hotter than the values, and values are accessed in a Zipfian way. HeMem thus manages to migrate some of the "hottest" pages to DRAM, but 45% of the memory accesses performed by Masstree still hit PMEM. In comparison, hardware caches operate at the cache line granularity, and the

hottest nodes and values are unlikely to conflict. On average, only 15% of the memory accesses hit PMEM with JC-static.

JC-dyn performs marginally better than JC-static because it detects that the pages used by the index are hotter than the pages used by the values. The difference with JC-static is negligible (14% of the data found in PMEM vs 15%).

It may seem surprising that migrations do not improve performance and that statically minimizing conflicts is enough to achieve close to optimal performance in Zipfian workloads, but conflicts between the hottest items are extremely unlikely (items are only 128B each in a 48GB cache). The benefit of adding active monitoring and conflict avoidance is thus negligible on average.

**Latency** The migrations performed by HeMem and JC-dyn have an impact on the observed latencies. Table 3 summarizes the latency spikes observed while running YCSB. While all systems have excellent 99p tail latency, the migration daemon pre-empts the Masstree threads, sometimes delaying the processing of a request by up to 4ms. Even though we use fewer threads than cores, the threads are not pinned to cores. The scheduler sometimes schedules two threads on the same core, explaining the pre-emption delays. The phenomenon happens when the scheduler tries to schedule threads that frequently block and unblock, such as the migration daemon.

Configuration	99p	Maximum latency
Linux	10us	10us
HeMem	10us	4ms
JC-static	10us	10us
JC-dyn	10us	4ms

Table 3: Maximum latency observed on the YCSB workload.

### 5.4.2 Performance over time, impact of the sampling rate

Both JC and HeMem perform better after a warm-up period: the DRAM cache needs time to cache accessed data, and HeMem needs time to detect and migrate hot pages to DRAM. Figure 9 presents the evolution of the performance of YCSB C. We initialize Masstree by inserting keys in random order, and then launch multiple iterations of YCSB C. Each iteration of YCSB C performs 10 million lookups, and keys



are accessed following a Zipfian distribution. For HeMem, we compare 4 configurations with varying sampling rates. *HeMem-1K* corresponds to the highest sampling rate, with 1 sample analyzed every 1,000 memory accesses, and *HeMem-50K* to the lowest sampling rate. The default rate of HeMem is *HeMem-10K*. We also evaluate the impact of the sampling rate on JC-dyn (*JC-10K* and *JC-50K*).

Figure 9 illustrates the impact of the sampling rate on performance. When the sampling rate is too high, the overhead of sampling negatively impacts performance (*HeMem-1K*). Even when the profiling and migration threads run on dedicated cores, the other cores still handle the interrupts generated by the performance monitoring units of the CPU when a memory access is sampled. These interrupts explain the lower performance of HeMem-1K. When the sampling rate is too low, many accessed pages are never marked as hot and are never migrated to DRAM (*HeMem-50K*). In this benchmark, the optimal performance of HeMem is reached when the sampling rate is close to the default sampling rate (*HeMem-5K*, *HeMem-10K*).

JC is less impacted by such considerations because its performance is good even without any conflict avoidance daemon. JC-dyn (*JC-50K*) also fails to detect any conflicts, but its performance reaches 3% of our optimal configuration after 100s of execution. Even without any conflict avoidance daemon (*JC-static*), JC is only 5% slower than the optimal configuration.

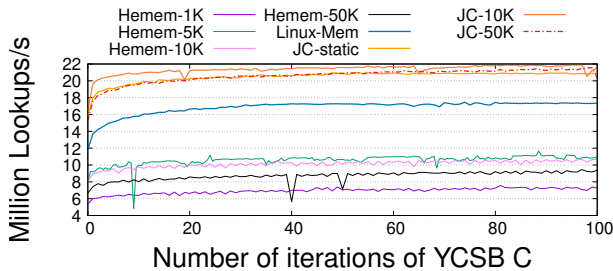


Figure 9: Performance of YCSB C. Every iteration of YCSB C runs 10 million queries.

### 5.4.3 Summary

Hardware caches outperform page-based migrations when working with scattered small items. Again, caches tend to "work well" when conflicts are minimized at allocation time, and their performance is not strongly dependent on monitoring memory accesses to find and fix possible conflicts.

## 5.5 Silo

Silo is configured to execute a TPC-C workload on a 100GB database. The TPC-C workload is heavily skewed: most of

the TPC-C data consists of the description of (sold) items, but most of the memory accesses are done on the customer and warehouse metadata. Figure 10 summarizes the performance of Silo, varying the number of threads.

Due to the order of initialization of the database, most of the hot working set used by Silo is allocated at the beginning of the execution. JC is able to allocate hot pages in a non-conflicting way, and HeMem allocates most of the hot pages in DRAM. Both JC and HeMem perform equally well on this workload, but better than Linux.

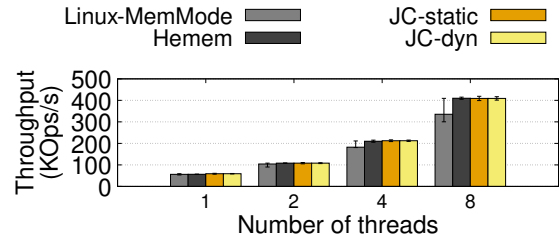


Figure 10: SILO (TPC-C) throughput (higher is better)

## 5.6 NAS benchmarks

Figure 11 presents the performance of the NAS benchmark suite running with Linux, HeMem and JC. We only include applications that executed in less than 24 hours on our machine.

Most HPC applications follow the same pattern as BC: large arrays are allocated and initialized at the beginning of the application, and then only a subset of the arrays is used during the execution of the algorithm. When the hot arrays fit in the DRAM cache, JC and Linux outperform HeMem by up to 2.8 $\times$  (class D size of the NAS benchmark, on the left of Figure 11). As BC, the NAS applications use OpenMP to parallelize their computation, and the profiling and migration threads of JC-dyn and HeMem have cascading effects on the performance of threads waiting at barriers.

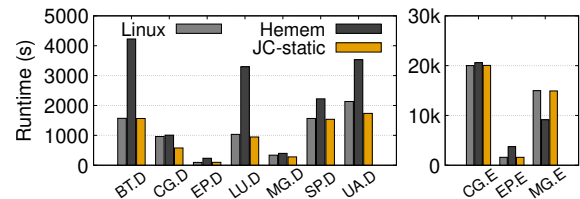


Figure 11: NAS application runtime (lower is better). JC outperforms Linux and HeMem except when the hot set size vastly exceeds the cache size (CG.E, MG.E).

The NAS benchmarks also allow us to demonstrate the limitations of our approach. On MG.E, HeMem runs 1.8 $\times$  faster than JC, despite its profiling overhead. MG uniformly accesses a large array and does not benefit from DRAM caching:

most of the cached data is evicted before being reused. It is well known that applications with uniform access to large data sets or streaming access patterns do not benefit from caching. For instance, in large streaming workloads, the streamed content keeps replacing itself in the cache, and data is always evicted before being re-read. In the worst case scenario, with DRAM caching, 100% of the memory accesses end up in PMEM. With software-based migration, some of the data is allocated in DRAM, and some of the memory accesses are resolved in DRAM.

Such applications are hard to support efficiently at the kernel level because no data is hot, and no conflict is particularly worthy of fixing. However, caching could be improved at the hardware level. CPUs already implement special cases for streaming workloads in their CPU caches: most recent CPUs implement QLRU, in which data that was cached by a streaming thread is evicted before data cached by threads performing random accesses [7]. Such a strategy could be implemented in a DRAM cache as well, for instance by avoiding caching large streams. The performance of the DRAM cache could also be improved by optimistically flushing dirty data to PMEM, when PMEM is idle, to reduce the latency of future evictions of dirty data.

**Summary** JC-static equals or outperforms Linux and HeMem on most NAS benchmarks. When, however, an application does not have a clear hot dataset, but rather streams or accesses large datasets that do not fit in the DRAM cache, DRAM caches are inferior to software migration.

## 6 Discussion

**Recommendations** From our experience, working with hardware caches and page migration systems, no solution fits all workloads, but the general rule of thumb is:

- Systems that rely on monitoring memory accesses are finicky to configure and can introduce huge performance overheads if not properly fine-tuned. In our experience, it is more likely for a migration daemon to be misconfigured than to perform well. This observation is not unique to this paper nor to the monitoring done by HeMem and JC-dyn. For instance, by default, most Linux distributions deactivate AutoNUMA, the page migration daemon of Linux because it negatively impacts most workloads. So, unless working with a known and predictable workload, we recommend using hardware caches with a static page allocation policy.
- When working with very large datasets that do not have a clear hot subset, caches should be avoided.

A surprising observation of this paper is that, for many workloads, large hardware caches perform close to optimally with a static page allocation policy, and that having a conflict

avoidance daemon is unnecessary. This seemingly counter-intuitive observation is explained as follows: conflicts that would be fixed by a daemon happen between frequently accessed cache lines. The number of such cache lines has to be small compared to the size of the DRAM: at current DRAM speed, it takes a few seconds to read the full DRAM cache, so any dataset that is large compared to the DRAM cache size cannot be "frequently" accessed. Because the number of frequently accessed cache lines is small compared to the DRAM size, the likelihood of problematic conflicts is small and a conflict avoidance daemon is more often a source of overheads than useful.

It is possible to craft adversarial workloads for which the static page allocation policy underperforms, and in which the dynamic policy performs well. In hand-crafted corner-case workloads, we found that running the conflict avoidance daemon infrequently and with a low sampling rate was enough to detect the most problematic conflicts and get close-to-optimal performance.

**Applicability to systems other than DRAM+PMEM** To the best of our knowledge, Intel's Memory Mode is the only currently commercially available hardware DRAM cache, so we focused the performance evaluation on DRAM+PMEM systems. We believe that the findings of this paper apply more broadly. Indeed, we have shown that tracking memory accesses at the software level is costly (profiling overhead) and requires migrating a large amount of data (migration overhead). These observations are fundamental limitations of software migration and independent of the underlying technology. If anything, software migration cost is likely to increase in future hardware with larger and faster memory – higher sampling rates will be required to detect and migrate more pages faster, incurring even more CPU overhead.

In comparison, provided that conflicts are minimized, hardware caches tend to "work well by default". Because hardware caches perform close to optimally even without any active conflict avoidance daemon, they can be operated with limited or no CPU overhead, and are more likely to perform well on future hardware.

## 7 Related Work

**Software-managed migration** Previous work focused on managing tiered memory systems at the software level. HeMem [20] is the state-of-the-art page migration system for DRAM+PMEM systems. HeMem focused on reducing the overhead of page migration, but still suffers from profiling and metadata overheads. Over the years, multiple metrics have been explored to accurately infer the heat of pages. Thermostat [1] and AutoNUMA [9] compute heat by sampling the accessed bit of the page table. Nimble [25] uses the OS active/inactive page list. TMO [23] counts the number of cycles wasted waiting for unavailable resources. HeteroOS [14]

uses hints from guest OSes to help the host OS perform informed page placement decisions. X-Mem [12] uses hints from the application developers to compute the hottest pages. UniMEM [24] uses performance counters and hints from the MPI runtime. Carrefour [11] gathers high-level performance metrics from the CPU (e.g., average latency of memory accesses) to tune the frequency of memory access sampling. All these works show that measuring heat accurately is a hard problem, but crucial to the performance of software migration. In this paper, we have shown that hardware caches are less sensitive to heat measurement and can even operate efficiently without if conflicts are statically minimized at page allocation time.

**CPU cache management systems** In this work, we assume the DRAM cache to be a 1-way directly mapped cache. This assumption holds true on current systems, and is likely to hold true in the future – for large caches DRAM, 1-way caches have been shown to outperform multi-way caches [18].

Multiple strategies have been proposed for maximizing the efficiency of CPU caches. In the early 90s, Kessler et al. [15] simulated the relationship between page placement and conflicts in caches and showed that it is possible to reduce the number of conflicts at allocation time. Bershad et al. [5] simulated the impact of page migration on the efficiency of caches. The generalization of caches with large associativity (many-ways CPU caches) allowed CPUs to keep a few conflicting cache lines in their caches and reduced the impact of system-level page placement on the performance of caches. These techniques have gradually been replaced by much coarser-grain page coloring techniques that partition the cache to avoid cache trashing between users or applications [6, 26] or by scheduling techniques to better share the cache between cache-intensive and cache-friendly applications [2, 21, 27]. It is interesting to note that current DRAM caches resemble the state of large CPU caches simulated in the 90s, and that page allocation policies matter in current tiered memory systems. In this work, we chose to avoid partitioning the cache. Adding page coloring on top of conflict minimization could be implemented to give a larger portion of the DRAM cache to an application.

The impact of page placement on cache performance has also been studied on Intel Xeon Phis. Xeon Phis can be configured to use a large MCDRAM pool as a hardware cache that sits in front of DRAM. Intel’s Zonesort [28] aims at limiting conflicts in the MCDRAM pool at page allocation time. In its first release, ZoneSort [10] periodically sorted the list of available free pages in an order that limits conflicts with already allocated pages. The module incurred significant CPU overhead and only partially limited conflicts. A later version of ZoneSort [28] allocated pages from bins in a round-robin order, an approach which does not always minimize conflicts when pages are not freed in the same order as they are allocated. JC always allocates pages from the bin with the

lowest heat. Zonesort was thought of as a temporary solution for applications that have not been adapted to the Xeon Phi architecture. In our paper, we show that the hardware management of a tiered memory system, combined with low-overhead conflict avoidance techniques, outperforms traditional page migration on a wide range of workloads. We believe that this novel counter-intuitive conclusion is important in the widening context of cacheable disaggregated memory.

## 8 Conclusion

We have demonstrated that hardware caches offer better performance than software management of tiered main memory systems, provided minor modifications of the operating system. We have shown that, surprisingly, statically minimizing conflicts at allocation time is sufficient to avoid most conflicts between hot pages in the cache.

**Acknowledgements.** We would like to thank our shepherd, Emery Berger, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Australian Research Council Grant DP210101984.

## References

- [1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.
- [2] Reza Azimi, David K Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multi-core processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, 2009.
- [3] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.
- [4] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.
- [5] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 158–170, 1994.

- [6] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices*, 31(9):244–255, 1996.
- [7] Zixian Cai, Stephen M Blackburn, and Michael D Bond. Understanding and utilizing hardware transactional memory capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 1–14, 2021.
- [8] Many contributors. Samsung Electronics Introduces Industry’s First 512GB CXL Memory Module. "<https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module>", 2022.
- [9] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. "<https://lwn.net/Articles/488709/>", 2019.
- [10] Intel Corporation. ZoneSort module. "[https://github.com/oslab-swrc/flsched/blob/main/knc/linux/drivers/zonesort/zonesort\\_module.c](https://github.com/oslab-swrc/flsched/blob/main/knc/linux/drivers/zonesort/zonesort_module.c)", 2017.
- [11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.
- [12] Subramanya R Dullloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
- [13] Intel. How Does the DRAM Caching Work in Memory Mode Using Intel® Optane™ Persistent Memory? "<https://www.intel.com/content/www/us/en/support/articles/000055901/memory-and-storage/intel-optane-persistent-memory.html>", 2021.
- [14] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 521–534, 2017.
- [15] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.
- [16] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.
- [17] Amar Phanishayee, David G Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flexkv: Enabling high-performance and flexible kv systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24, 2012.
- [18] Moinuddin K Qureshi and Gabe H Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246. IEEE, 2012.
- [19] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem - artifact. "<https://sysartifacts.github.io/sosp2021/results.html>", 2021.
- [20] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.
- [21] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *ACM SIGOPS Operating Systems Review*, 41(3):47–58, 2007.
- [22] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.
- [23] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.
- [24] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.
- [25] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered



memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

- [26] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.
- [27] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.
- [28] Daniluk Łukasz. mm: Add cache coloring mechanism. "<https://lkml.org/lkml/2017/8/23/195>", 2017.

# TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers

Amogha Udupa Shankaranarayana Gopal   Raveendra Soori   Michael Ferdman   Dongyoon Lee

*Stony Brook University*

## Abstract

A heap overflow vulnerability occurs when a program written in an unmanaged language such as C or C++ accesses a memory location beyond an object allocation boundary. Malicious users may exploit this vulnerability to corrupt an adjacent object in memory, creating an entry point for a security attack. Despite decades of research, unfortunately, it still remains challenging to detect heap overflow vulnerabilities in real-world programs at a low cost.

We present TAILCHECK, a new lightweight heap overflow detection scheme that leverages page protection and pointer tagging. When an object is created, TAILCHECK allocates an additional page-protected shadow object, called a TailObject, placing the distance from the object to its TailObject as a tag stored in the unused high-order bits of the object pointer. For every access to the original object, TAILCHECK performs an additional memory access to the TailObject, whose address is computed using the tag. Heap overflows are detected as page faults when an access occurs beyond the TailObject. We evaluated TAILCHECK with four server applications (apache, nginx, memcached, redis) and the SPEC CPU2017 and SPEC CPU2006 benchmarks, successfully finding heap overflows in SPEC CPU2017 gcc. TAILCHECK experiences 4% and 3% run-time overhead for the average and tail (99%) latencies for server applications; and only 33% and 29% run-time overhead for SPEC CPU2017 and SPEC CPU2006, respectively, less than the state-of-the-art solution.

## 1 Introduction

A heap overflow [47,48] is an anomaly that occurs when a program attempts to access a memory location beyond the bounds of its allocated memory. This type of vulnerability is commonly found in programs written in unmanaged languages such as C and C++, as these languages allow programmers to directly manipulate pointers without providing compile-time (*e.g.*, as in Rust) or run-time protection (*e.g.*, as in Java or Go). A malicious user may exploit a heap overflow vulnerability in a C or C++ program to perform a variety of security

attacks [57,58], including corrupting code pointers to divert control flow or leaking sensitive information.

Today, many critical software systems—such as server applications and operating systems—are developed in unsafe languages. Programming errors in these systems can therefore lead to heap overflow exploitation. For example, a vulnerability in the `nginx` web server (CVE-2014-0133) allowed attackers to send a specially-crafted request that caused a heap overflow, allowing them to execute arbitrary code on the server. The `mysql` database code had a vulnerability (CVE-2021-2429) which allowed attackers to send a specially-crafted request that caused a heap overflow, potentially gaining access to the data or taking control of the database. A heap overflow in the PHP programming language related to encryption (CVE-2022-37454) could be used to remotely execute arbitrary code on a web server. The prevalence of heap overflow vulnerabilities in deployed software systems highlights the need for effective run-time techniques to protect production systems against exploitation, even when running vulnerable software.

Several systems have made significant strides toward run-time mitigation of heap overflows. AddressSanitizer [54], the state-of-the-practice solution, incurs high run-time overhead: 80% (geometric mean) slowdown for SPEC CPU2006 applications [45]. Modern operating systems offer heap overflow protection by allocating an object at the boundary of a virtual memory page and adding a protected page (with no access permission) after it; this feature is available in Linux as Electric Fence [49] and in Windows as PageHeap [61]. However, allocating just one object per protected page suffers from extremely large memory overhead, along with high run-time cost due to frequent TLB misses. Delta Pointers [28], the state-of-the-art technique, achieves the lowest run-time overhead (35% for SPEC CPU2006), but requires restricting the address space of the protected application. Delta Pointers reserves the  $N$  most significant bits (32, by default) for pointer tagging and supports only a  $48 - N$  bit address space for 64-bit architectures. This limits Delta Pointers' applicability for modern software: it cannot be used for server software

with many-gigabyte footprints and even fails for the reference inputs of `xz` and `mcf` in the SPEC CPU2017 suite.

In this work, we present TAILCHECK, a new lightweight heap overflow detection scheme that leverages a custom memory allocator, OS-based page protection, and compiler-directed pointer tagging. When an object is created, TAILCHECK allocates an additional shadow object, called a TailObject, at the boundary of a page whose subsequent page is protected by the OS. The TAILCHECK memory allocator returns a tagged pointer in which the otherwise unused most significant bits (e.g., 16 bits for a 64-bit architecture with 48-bit address space) encode the distance from the original object to its TailObject, keeping the address of the original object unmodified in the low-order bits as usual. A TAILCHECK compiler pass instruments each dereference of a tagged pointer, using the embedded tag to compute the shadow address within the corresponding TailObject and inserting an additional memory access to the shadow address alongside each access to the original object. In the event of a heap overflow, the shadow memory accesses reach beyond the bounds of the TailObject, causing a page fault and triggering the OS to terminate the program. This prevents the exploitation of heap overflow vulnerabilities (both over-writes and over-reads) and ensures the integrity and confidentiality of the system.

The TAILCHECK tags allow many objects to share space used by the TailObjects and the OS-protected pages, limiting the memory overhead of the technique and eliminating the performance overheads of frequent system calls to protect pages during memory allocation. To further reduce run-time overhead, TAILCHECK performs three compile-time optimizations to prune the shadow accesses for heap accesses that are statically proven to be safe.

TAILCHECK makes use of well-known page protection and pointer tagging techniques, yet it does not share the limitations of prior solutions. TAILCHECK achieves low run-time overhead by using page protection for heap overflow detection, but unlike Electric Fence and PageHeap, it allows multiple small objects to be co-located on a virtual memory page. TAILCHECK uses pointer tagging, but unlike Delta Pointers, it allows a program to utilize the full address space by only re-purposing the otherwise unused most significant bits.

We implemented TAILCHECK by extending the *mimalloc* allocator [33] and developing LLVM [31] compiler passes for code instrumentation. We evaluated TAILCHECK with four server applications (`apache`, `nginx`, `memcached`, `redis`) and the SPEC CPU2017 and SPEC CPU2006 benchmark suites. Interestingly, TAILCHECK identified an out-of-bounds read in SPEC CPU2017 `gcc` (v4.5.0), a known bug with an available patch [1], yet the patch is not present in SPEC CPU2017 v1.0.5. For performance, TAILCHECK experiences 4% and 3% run-time overhead for the average and 99% tail latencies for server applications. TAILCHECK exhibits 33% (geometric mean) run-time overhead and 3% memory overhead for SPEC CPU2017. TAILCHECK exhibits 29% (geometric mean) run-

time overhead for SPEC CPU2006, lower than Delta Pointers, the state-of-the-art compiler-based solution with the lowest previously-reported run-time overhead (35%).

This paper makes the following contributions:

- To the best of our knowledge, TAILCHECK is the first lightweight heap overflow detection scheme based on page protection that does not place one object per page.
- TAILCHECK introduces a new pointer tagging scheme for heap overflow detection, which encodes distance metadata only in the otherwise unused pointer bits and thus does not restrict the application address space.
- An evaluation of TAILCHECK demonstrates that it incurs low run-time and memory overheads and supports applications with large many-gigabyte memory requirements.

## 2 Background & Motivation

This section briefly describes the background on heap overflows, discusses the threat model we assume in this work, and highlights the need for a new solution.

### 2.1 A Heap Overflow Vulnerability

The lack of run-time and compile-time heap overflow protection in C and C++ exposes many critical software systems to security threats. Stack-based buffer overflows have received significant early attention from both academia and industry. Mature mitigations using stack canaries [11] and shadow stacks [60] are readily available: for example, GCC and Clang have built-in support with the `-fstack-protector` and `-fsanitize=safe-stack` compiler flags. On the contrary, standard solutions for heap overflows have not yet been settled, with solutions offering trade-offs in run-time and memory overheads, soundness, and completeness (§2.3).

Heap overflows are responsible for many critical real-world security problems. A heap overflow *over-write* is particularly critical as it may allow malicious users to divert the control flow of a victim program or gain privilege escalation. For instance, a heap overflow over-write vulnerability is found in `sudo` (CVE-2021-3156), a widely-used utility on Unix-like operating systems, which enables a user to run programs with the security privileges of another user. This heap overflow was particularly critical in that an attacker could control not only the size of the buffer that can be overflowed but also the size and contents of the overflow. As a result, when exploited, the vulnerability could allow an unprivileged malicious user to gain root privileges on a vulnerable host.

A heap overflow *over-read* can lead to information leakage. The Heartbleed [19] vulnerability in the popular OpenSSL cryptographic software library (CVE-2014-0160) is a representative example. A missing bounds check in the SSL/TLS heartbeat extension could be exploited to reveal up to 64KB of memory, which may include private keys and other secrets.

## 2.2 Threat Model

In this work, we address the threat of overflows on heap-allocated objects. We assume an attacker can feed a crafted malicious input to a victim program to exploit a heap overflow vulnerability. We mitigate heap overflows (both over-write and over-read) that occur in application and library code that can be instrumented with our tool, providing integrity and confidentiality when the underlying software contains vulnerable code. We provide no protection for uninstrumented code such as third-party libraries. We do not consider other memory safety violations such as use-after-free or uninitialized read vulnerabilities.

## 2.3 Motivation: Haven't we solved it yet?

Given the critical implication for security, many solutions have been proposed for mitigating heap overflows. We present several representative works, discussing their limitations and the lessons that we draw upon when designing TAILCHECK. A more comprehensive related work discussion follows in §8.

The idea of leveraging a virtual memory *protected page* to detect a heap overflow dates back to 1987. Electric Fence [49], proposed by Perens and now included in Linux, was the first to place allocated objects immediately before protected pages, which are configured by the OS to trigger a hardware page fault when accessed. Reads or writes beyond the allocated object would land on the protected page, triggering a fault and allowing the OS to mitigate the heap overflow. Successors to Electric Fence, including DUMA [5], DYBOC [56], OSX's libmalloc [35], and Windows' PageHeap [61] follow a similar design. However, despite its simplicity, the approach of allocating one heap object per virtual memory page has unacceptable memory overhead. Moreover, this approach incurs large run-time overheads from multiple sources: performing system calls to protect a page on every heap allocation is expensive, spreading heap allocations across many pages results in excessive TLB misses, and placing objects at common offsets from the end of memory pages increases data cache contention. For example, Liu *et al.* [36] reports Electric Fence incurs a 7x slowdown for the PARSEC benchmarks [7]. As a result, the idea of using protected pages for run-time heap overflow mitigation has lost its attraction and is rarely found outside of debugging environments. Using protected pages can offer heap overflow protection without explicit metadata lookups and bounds checks, yet require a new solution that supports placing multiple objects per virtual memory page and avoids frequent page protection system calls.

AddressSanitizer (ASan) [54] is an alternative approach with lower run-time overhead. ASan manages a fine-grained inaccessible region called a *redzone* after each allocated object by maintaining a disjoint shadow (metadata) memory space. On each memory access, ASan looks up the metadata space and checks if the target location falls in a redzone. Other

prior solutions, notably SoftBound [40], keep *base and bound* metadata in a shadow memory space. On each memory access, SoftBound performs a metadata lookup and explicitly checks that the instrumented access falls within the object bounds. Although the implementation details differ across these systems, they all share two downsides. First, the metadata lookup (comprising additional shift, add, and load instructions) and bounds check (including comparison and branching instructions) have significant run-time overhead. Second, the redzones and metadata store incur significant space overheads. For example, for the SPEC CPU2006 benchmarks, Oleksenko *et al.* [45] report 1.8x run-time and 4x memory overheads for ASan, and 2x run-time and 3x memory overheads for SoftBound. Such performance overheads and memory capacity requirements are unacceptably high for modern large-memory performance critical applications which would most benefit from run-time heap overflow protection.

To reduce the metadata overheads and/or to facilitate metadata lookup, researchers proposed *pointer tagging* techniques that keep metadata in the high-order bits of a pointer itself (*e.g.*, unused 16 bits in 64-bit architecture). For instance, Baggy Bounds [2] tags a pointer with the encoded size of an object. However, Baggy Bounds still requires expensive array table look-ups and bounds checking on pointer arithmetic. Taking one step further, Delta Pointers [28] remove the bounds check (comparison and branching instructions) by transforming the heap overflow detection problem into an integer overflow detection problem, managing pointer tags in a way that would cause out-of-bound pointer arithmetic to set an overflow bit in the tags, thereby making such pointers “uncanonical” and causing the MMU to generate a fault on dereference. Delta Pointers are considered the state-of-the-art software-only solution based on its low run-time overhead, exhibiting only 35% slowdown for SPEC CPU2006 benchmarks. However, Delta Pointer tags must include the distance from the current pointer to the end of an object, requiring significantly more than 16 bits for large objects. To work around this limitation, Delta Pointers shrink the process address space. By default, Delta Pointers use a 32-32 bit split, supporting applications with up to 4GB of address space. Delta Pointers offer a glimpse of a low-overhead solution without metadata lookups or bounds checks, but are still limited in terms of practicality due to its address space restrictions.

## 3 TAILCHECK Design

TAILCHECK extends the memory allocator and compiler to produce executables that are protected against heap overflow exploitation at run-time. Whenever a programming error leads to an access that overruns the end of a heap-allocated object, TAILCHECK ensures that a hardware page fault is triggered, causing the operating system to mitigate a potential attack by trapping the fault. Although this effect can be achieved by placing each heap object at the end of its own virtual memory



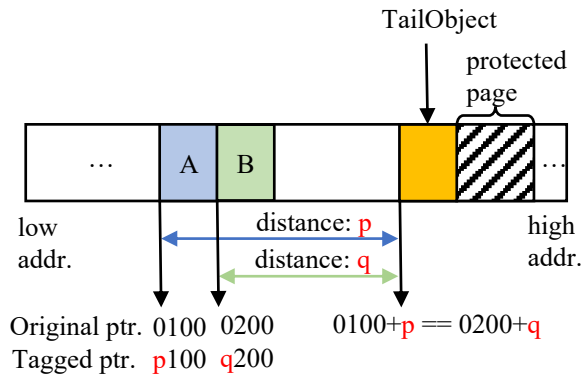


Figure 1: TAILCHECK allocates a (shadow) TailObject at the boundary of a protected page and tags object pointers with the distance between the original and shadow objects.

page (followed by a protected page) [49], such an approach is prohibitively expensive; allocating a virtual memory page for each heap object results in massive heap capacity bloat, while performance is severely impacted at the time of allocation (due to configuring a protected page on each allocation) and also at the time of use of the allocated object (due to a significant increase in TLB and cache pressure).

Rather than coupling each heap object with its own protected page, TAILCHECK reserves one TailObject and a protected page per memory region managed by the heap allocator. Figure 1 depicts this arrangement. When the TAILCHECK memory allocator requests a region of memory from the OS, it reserves space for the TailObject and configures a protected page at the end of the region. After this, the allocator places dynamically allocated heap objects, such as A and B, as usual.

To trigger a page fault on a heap overflow, the TAILCHECK compiler instruments the application code to perform a shadow memory access to the TailObject alongside each load and store operation to the original heap object. Figure 1 shows that the base address of object A is a known distance  $p$  away from the base address of its TailObject. The base address of object B is similarly a known distance  $q$  from its TailObject. To compute the address for the shadow memory access, the compiler simply adds the offset ( $p$  or  $q$  in this example) to the address of the original access, as shown in Figure 2. Although an access beyond object A (e.g., to the address  $p210$ ) would erroneously read or write data belonging to object B, the corresponding shadow access (e.g., to the address  $0210+p$ ) that TAILCHECK performs before the original object access will exceed the bounds of the TailObject, attempting to access the protected page and triggering a page fault.

To store the distances (such as  $p$  and  $q$ ) from the moment when heap objects are allocated to the time when they are accessed, TAILCHECK uses tagged pointers. We leverage the otherwise unused high-order bits of pointers to store the distances, using the compiler to emit code that masks these high-

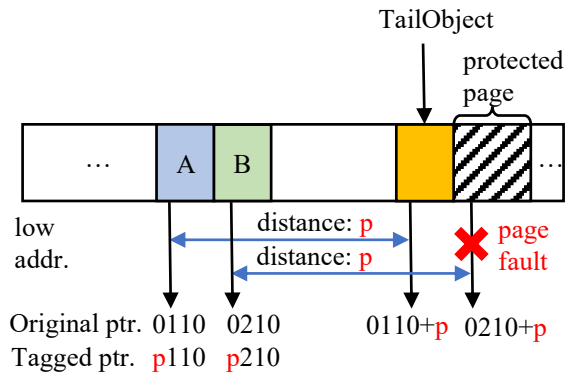


Figure 2: TAILCHECK adds shadow memory access for the TailObject. A heap overflow beyond object A is detected as a page fault on the shadow access beyond its TailObject.

order bits before performing an access to the original object and adds the distance encoded in these bits to compute the address of the shadow access. Modern x86 and ARM systems use a 48-bit address space, leaving 16 unused bits in 64-bit pointers, thereby allowing TAILCHECK to store distances for allocator regions of up to 64KB. A key advantage of this approach is that the distances encoded in the tagged pointers are propagated implicitly, requiring code instrumentation only at the time of pointer dereference or comparison. Notably, TAILCHECK still protects large heap objects allocated within their own allocator regions by using protected pages and setting the distance for shadow accesses to 0, therefore treating all accesses uniformly, but incurring a small overhead due to the duplication of memory accesses to the large objects.

We note that reserving space for the TailObjects is necessary because shadow memory accesses for stores write data into the TailObject space. Although these data are never used, if the space were not reserved, the application could allocate an unrelated object in the same space (at the end of the managed region), causing the shadow stores to clobber that object. An alternative implementation, which uses shadow loads for the corresponding original object stores, is possible. At first glance, this arrangement would eliminate the memory overheads of TAILCHECK. However, this approach can have significant performance drawbacks because loads are on the critical path of the processor pipeline and have a higher execution cost compared to stores.

### 3.1 TAILCHECK Code Instrumentation

We use the compiler to add TAILCHECK to executables. The compiler performs three tasks: it replaces memory allocation calls with the TAILCHECK allocator, adds shadow memory accesses at pointer dereference sites, and masks pointer tags when interacting with uninstrumented code (e.g., shared libraries). We detail these tasks below.

**Allocator Injection.** TAILCHECK uses a custom memory allocator. Unlike regular allocator functions that return a pointer carrying the address of the allocation in virtual memory, the TAILCHECK allocator functions return a tagged pointer. The tag is inserted into the otherwise unused high-order bits of the returned value, and corresponds to the distance between the address of the allocated heap object and the address of its corresponding TailObject.

Replacing the memory allocator provided by the standard library is traditionally done with the dynamic linker. However, TAILCHECK needs its custom allocator only for the heap objects that it protects. To inject its custom allocator during code instrumentation, the TAILCHECK compiler identifies allocator calls (*e.g.*, `malloc`, `new`, `strdup`, etc.) and replaces them with their TAILCHECK allocator equivalents. Any code linked into the executable, but not instrumented with TAILCHECK, continues to use the unmodified system allocator.

For each allocator-managed memory region requested from the OS by the TAILCHECK allocator, we reserve space for the TailObjects at the end of the region and `mprotect` the virtual memory page immediately following the region. TAILCHECK requires for the TailObject reservation at the end of the managed memory region to be as large as the largest object allocated within that region. In practice, it is common for modern allocators to separate memory regions by *size class*, dividing each region into slots of this size and allocating one object per slot. With this strategy, the TAILCHECK allocator can simply reserve the last slot within each managed memory region. Notably, the distance encoded in the pointer tags is computed for the TailObject address equal to *address\_of\_protected\_page minus size\_of\_allocated\_object*, which lands in the middle of the reserved slot whenever the allocated object size is smaller than the size class, or lands at the start of the reserved slot when the allocated object size equals the size class.

When the requested allocation size is larger than the largest allocator size class (*e.g.*, one that requires multiple virtual memory pages), modern allocators switch to a large-object allocation mode where a separate memory region is requested from the OS. When handling large-object requests, the TAILCHECK allocator will `mprotect` a page immediately following each allocated large object and compute the returned pointer as *address\_of\_protected\_page minus size\_of\_allocated\_object*, effectively falling back to the behavior of Electric Fence. For such large-object allocations, the tag of the returned pointer is set to 0.

We note that, when computing TailObject addresses, we round up the *size\_of\_allocated\_object* to honor the original object's memory alignment requirements. This is useful for both correctness and performance, as we want the shadow access instructions to have the same alignment properties as their corresponding original object access instructions. As a result of this rounding, objects whose requested size is not a multiple of their alignment size will have a small (several bytes) region where heap overflows may go undetected

with our TAILCHECK implementation. We discuss the generally benign nature of such overflows and the ramifications of adding support for precise overflow detection independent of alignment constraints in §7.2.

**Memory Access Instrumentation.** The TAILCHECK compiler operates at *module* granularity, treating all pointers local to a module as tagged pointers. On every pointer dereference in the instrumented code, a tagged pointer must be deconstructed into two parts, the object address (by masking the tag) and the shadow access address (by adding the tag to the object address). A compiler pass iterates over the pointer dereferences, inserting compiler IR for tag handling and injecting the TailObject shadow accesses. Each shadow access (load or store) is performed first, immediately followed by the original object access. For both loads and stores, the same store value and load target registers can be used by the two accesses, avoiding tying up additional register resources for the shadow accesses. Compiler optimization passes are performed both before and after the TAILCHECK instrumentation pass, ensuring that all dereferences eliminated by the optimizer are not instrumented and that the address calculation code for handling the tagged pointers is optimized. The shadow accesses are marked as volatile to ensure that they are not moved or eliminated as dead code.

Pointer arithmetic with integers does not require special handling for tagged pointers. However, whenever a tagged pointer is compared (either to another tagged pointer or to NULL), the TAILCHECK compiler must mask the tag bits prior to the comparison. Similar to the case of pointer dereference instrumentation, comparison tag manipulation logic is inserted as compiler IR, allowing an optimization pass to minimize the overhead of these operations.

We note that memory access instrumentation does not differentiate between objects allocated with the regular and large-object modes. All pointers within the instrumented code are treated as tagged pointers. For large-object allocations, the tags are set to 0 by the memory allocator, resulting in minor overhead for tag manipulation and harmless shadow accesses that load or store exactly the same address as the original access. Uniform handling of tagged pointers simplifies the implementation, while the overhead of back-to-back instructions that access the same cache line is minimal in modern superscalar out-of-order processors.

**Linking with Uninstrumented Code.** Although it is theoretically possible to compile all modules of a program with TAILCHECK instrumentation, in practice, we must provide the ability to link against uninstrumented modules, such as shared libraries. In these situations, pointers passed by instrumented code into uninstrumented code (*e.g.*, library function calls that have pointers in their arguments) must have the pointer tags removed. Even if all libraries, including the standard library, are instrumented with TAILCHECK, there are still

situations where pointers are passed as arguments to system calls, also requiring the stripping of tags.

The TAILCHECK compiler performs a pass over the instrumented module to identify all function call sites. Calls to functions within the same module receive unmodified tagged pointers as arguments. However, for calls to external functions, compiler IR is inserted to mask the tags of all pointer arguments. Notably, it is safe to pass function pointers of instrumented functions to uninstrumented code (e.g., as callbacks), as any pointer arguments passed by uninstrumented code into these functions will have tags set to 0 and will execute correctly, albeit with harmless duplicated memory accesses as in the case of the large-object allocations.

TAILCHECK keeps pointers without tags in globals because they may be accessed by uninstrumented libraries. To this end, TAILCHECK identifies all pointer stores to global variables in the instrumented code and ensures that the values written into these variables are masked prior to being stored. The TAILCHECK compiler uses the LLVM instruction operand type `GlobalVariable` to identify globals (after calling `stripPointerCasts` on the operand). There is a possibility that TAILCHECK may store a tagged pointer into a global if a program uses a local alias to write to that global, potentially leading uninstrumented code to later dereference a tagged pointer stored in the global. However, we observe that accessing a global variable via a local pointer alias is uncommon in practice: during our evaluation (§6), none of the tested server, SPEC CPU 2017, or SPEC CPU 2006 applications shows any unexpected behavior (e.g., segmentation fault), which would happen if uninstrumented libraries accessed tagged pointers in globals. Thus TAILCHECK does not perform additional pointer alias (provenance) analysis. Furthermore, we treat the `environ` variable as a special case where not only the variable itself, but also the nested pointers within its structure, are written with their tags masked.

Finally, if the standard library is not instrumented with TAILCHECK, there are two classes of commonly used functions (`mem*` and `str*`) that can benefit from special handling, following the practices of previous works [28, 29, 45]. These functions are often responsible for heap overflows, making it practical to insert bounds checks at their call sites when their function bodies (part of the standard library) are not instrumented. For the *MemIntrinsics* functions (`memcpy`, `memmove`, and `memset`), we inject a TAILCHECK-like bounds check by performing a shadow access to the last byte of the arrays passed as arguments. Notably, we must first check that the `size` argument is non-zero, as the function semantics dictate that no pointer dereferences occur if the size is zero. The common string manipulation functions (e.g., `strstr`, `strchr`, etc.) return pointers. Although instrumented code handles these functions correctly (treating them like other 0-tag pointers), calling these functions effectively removes TAILCHECK protection from the pointers passed to them. To avoid losing heap overflow protection after these function calls, TAILCHECK

instruments the call sites of these functions to save the tags of the pointer arguments before the call and re-applies the tags on the returned pointers. Similarly, TAILCHECK masks tags in the return values of those functions that convert a string to a number (e.g., `strtol`) when used in arithmetic operations.

**Mixing Memory Allocators** Having the same allocator in the application and shared libraries is not a requirement for TAILCHECK. In our setup, we use `LD_LIBRARY_PATH` to ensure that all linked libraries use the TAILCHECK memory allocator, primarily to maintain performance consistency across all experiments. However, it is worth noting that libraries cannot and should not call `free()` on objects they did not allocate themselves [53]. If application code includes such a construct, it would cause failures in many scenarios (e.g., where custom allocators are used), including with the TAILCHECK allocator.

**Custom Memory Allocators** By default, TAILCHECK protects heap objects that are allocated and deallocated via standard interfaces (e.g., `malloc`, `realloc`, and `free`), replaced by `LD_LIBRARY_PATH`. Thus, there could be a heap protection granularity mismatch if an application uses a custom allocator. For example, for an application-level pool (slab) allocator, TAILCHECK may protect a `malloc`-allocated pool at a coarse granularity, not at the fine-grained custom allocation granularity. Our `nginx` server evaluation (§6) compares two cases with and without application-level pool allocations (by disabling a pool allocator using a debugging flag).

## 3.2 TAILCHECK Optimizations

To reduce the performance overheads of TAILCHECK, we apply several compiler IR optimizations that reduce the cost of instrumentation. We detail these optimizations below.

**Merging Tag Handling.** The tags of TAILCHECK tagged pointers remain constant throughout the lifetime of the pointer. When the same pointer is dereferenced multiple times within a function, potentially with different offsets (for accessing different members of the heap object), the operations to compute the `TailObject` pointers are redundant. To reduce the overhead of this common case, the TAILCHECK compiler instrumentation pass keeps track of the computed `TailObject` pointers and reuses their already computed values.

**Hoisting Tag Handling.** Heap pointers are frequently dereferenced inside loops, accessing different locations within the same heap object (i.e., if the object is an array). To reduce the overhead of tag handling, we hoist the computation of the `TailObject` pointer outside of the loop, leaving only the dereference operations inside the loop body. As a result of this optimization, the `TailObject` accesses within the loop

body exactly mimic the original object accesses, including using the same x86 scale-index-base-displacement for the shadow memory access and pushing all other TAILCHECK instrumentation overheads outside of the loop body.

**Statically Safe Dereferences.** TAILCHECK is effective at preventing heap overflow exploitation with relatively low overheads at run-time. However, while many pointer dereferences must be verified (*e.g.*, using shadow accesses to the TailObjects), some of the checks are unnecessary because static analysis of the code can guarantee that all accesses remain within the bounds of a heap object. As such, to further reduce the overhead of TAILCHECK, we adopt the *SafeAllocs* [28] static analysis implementation from the Delta Pointers work.

SafeAllocs identifies all heap allocations with statically known sizes and uses the compiler metadata to track object bounds along with the pointer corresponding pointer. Whenever such pointers are dereferenced in the code, the compiler checks if the offset of the dereference can be statistically determined and, if it can be determined and falls within the object bounds, a run-time check is unnecessary.

When SafeAlloc indicates that all accesses to a heap object are known to be safe at compile time, TAILCHECK uses the standard memory allocator for these objects and does not introduce shadow accesses for them. Some heap objects have both dereference sites that are known to be safe and also dereference sites that must be checked at run-time. We statically identify the safe regions at function granularity, avoiding shadow accesses for objects whose accesses are known to be safe. This also requires masking the tag bits of these pointers in the function preambles, as these objects are still allocated using the TAILCHECK custom allocator and the function call sites continue to pass arguments as tagged pointers.

## 4 TAILCHECK Implementation Details

We develop the TAILCHECK prototype by extending the *mi-malloc* allocator [33] and developing LLVM [31] compiler passes for code instrumentation. Tagged pointers are returned by the TAILCHECK allocator for allocations up to 16KB, with all larger requests treated as large-object allocations.

The TAILCHECK instrumentation is performed using three compiler passes. First, a SafeAllocs pass is done to identify optimization opportunities. Then a Call-Site Instrumentation pass replaces memory allocation function calls (`malloc`, `calloc`, `realloc`, `strdup`, `strndup`, and `free`) with the TAILCHECK custom allocator versions of these functions and masks pointer arguments at call sites of external functions. The Dereference Instrumentation pass inserts shadow loads and stores to the TailObjects for all heap objects requiring run-time checks. These passes are performed as part of the link-time optimization, ensuring that all statically linked sub-modules are combined together into one module for

TAILCHECK instrumentation before the passes are performed. Standard LLVM compiler optimization passes are performed both before and after the TAILCHECK passes.

We take special care to handle function arguments with the `byval` attribute. In LLVM, the `byval` attribute at a call site means that the pointer must be dereferenced and the resulting value copied before being passed as an argument. Because the LLVM `byval` mechanisms cannot handle tagged pointers, we mask the tags of all pointers with the `byval` attribute.

All tagged pointer-based solutions present challenges when linking to uninstrumented libraries, as tagged pointers must be masked before being passed to functions in uninstrumented code. Although pointers to data structures have their tags masked at the function call sites by the Call-Site Instrumentation pass, the nested pointers within these data structures are written as tagged pointers by the TAILCHECK instrumented code and cannot be directly dereferenced by the uninstrumented functions. As in prior work [2, 8, 28], we assume that we can soundly enumerate all call sites of external uninstrumented functions that will operate on nested pointers, and inject the necessary instrumentation code to mask nested tagged pointers. Notably, most C++ Standard Template Library (STL) containers do not require masking of nested pointers because they are implemented in header files and thus come within our instrumentation scope. For the select cases we encountered in our benchmark applications that require masking, we manually add the appropriate instrumentation as discussed in §5. In §7.3 we discuss how TAILCHECK may take advantage of the ARM top-byte-ignore Memory Tagging Extension (MTE) [3] and similar features in other ISAs to avoid the need for explicitly masking pointer tags.

## 5 Evaluation Methodology

We conduct all experiments on a system with an Intel Xeon Gold 5218 CPU. To benchmark TAILCHECK, we use four popular server applications (*apache* v2.4.54, *nginx* v1.22.1, *memcached* v1.6.17, *redis* v7.0.6), as well as the C and C++ applications from the SPEC CPU2017 and SPEC CPU2006 benchmark suites. For SPEC CPU2017, we use the *speed* set and limit applications to one thread.

Server applications often have a custom pool-based allocator. To evaluate potential performance differences between coarse-grained and fine-grained memory allocations, in addition to the *nginx* server results, we also present “*nginx (w/o poolalloc)*,” which is compiled with the `-DNGX_DEBUG_PALLOC=1` flag to disable its custom pool allocator and to use `malloc` and `free` directly. *apache* (v2.4.54) and *memcached* (v1.6.17) do not provide similar pool allocation on/off options, while *redis* does not use pool allocation.

To quantify the performance of the web servers *apache*, *nginx*, and *nginx (w/o poolalloc)*, we measure request latency using the `hey` HTTP load generator [16]. We create two workers to repeatedly request a file 256 times per second. We test



four different file sizes: 32KB, 128KB, 512KB, and 2MB. We configure `apache` with two worker threads and `nginx` with one worker process. For the key value stores, `memcached` and `redis`, we measure the request latency with four workers, each requesting 128,000 keys with a 50% get/set ratio. We use a key size of 16 bytes and four different object sizes: 32B, 128B, 512B, and 2KB. For the SPEC CPU2017 and SPEC CPU2006 benchmarks, we measure performance with reference input as wall-clock time of program execution.

For a fair comparison across all systems, we use unmodified `mimalloc` [33] for all evaluated configurations except `TAILCHECK`. For `TAILCHECK`, we use unmodified `mimalloc` for the uninstrumented code and only extend the `mimalloc` functionality with wrappers for the allocation functions, retaining all of the core functionality of the `mimalloc` allocator even when called from the instrumented code. The memory overheads we report are measured as peak resident set size.

As part of our evaluation, we include a comparison to Delta Pointers [28] and AddressSanitizer [54]. To ensure fairness, we reproduce the Delta Pointers results in our test environment after enabling only the comparable heap overflow protection features and ensuring that all available optimizations are applied. To make the results directly comparable, we perform this study with the same SPEC CPU2006 benchmark suite that was used in the original Delta Pointers publication. AddressSanitizer is compared for the server applications.

`TAILCHECK` works for all SPEC CPU 2017 benchmarks using LLVM -O3 with Link Time Optimization (LTO). However, we use -O2 and LTO in our evaluation to make the results directly comparable to the prior work [28]. We introduced specialized handling for the following benchmark applications to address compatibility issues with uninstrumented libraries:

- In the case of `403.gcc`, pointers stored in “long long” variables are passed to functions invoked through function pointers. Consequently, an uninstrumented `libc` function is called with a long long argument containing a tagged pointer. This causes a segmentation fault in the uninstrumented code, with the faulting address being a tagged pointer. Debugging this situation is straightforward, as the stack trace directly points to the problem. To address this, we utilized source instrumentation and manual pointer tag masking in the benchmark sources, similar to techniques applied in previous works [28, 29, 45].
- `520.omnetpp` employs a C++ data structure, `evbuf`, inherited from `basic_stringbuf`. This object contains a nested tagged pointer, whose information is lost due to C++ inheritance, leading to a tagged pointer being passed to a `libstdc++` function. This triggers a segmentation fault, easily identified by the faulting tagged pointer. To overcome this, we explicitly marked the `evbuf` type to ensure its members are always written as untagged pointers, thereby maintaining the integrity of the passed pointer irrespective of inheritance nuances.

Except for the above two cases, `TAILCHECK` is compatible with many complex real-world applications, including four servers and all other SPEC CPU 2017 and SPEC CPU 2006 applications. We note that although the two exception cases were easily identifiable and debuggable because they triggered a segmentation fault, it is theoretically possible that a tagged pointer may lead to silent data corruption and exhibit an observable event far later in time. `TAILCHECK` provides limited support for such cases, and fixing them may require manual code reviews. Indeed, addressing compatibility issues with uninstrumented libraries is a common limitation of pointer tagging-based solutions [2, 8, 28] with a notable exception `LowFat` [30] (see related work discussion in §8.2).

## 6 Evaluation Results

Below, we first describe the heap overflow vulnerabilities that were successfully caught by `TAILCHECK`. We then present the performance and memory overheads of our technique, and explain the impact of the optimizations described in §3.2 which mitigate some of the performance impacts. Finally, we present a comparison with the prior art, demonstrating comparable and lower overheads compared to Delta Pointers, without being subject to its address space limitations.

### 6.1 Heap Overflow Detection

We developed a set of test cases that exhibit various types of heap overflows to ensure that a segmentation fault is experienced when running such cases when the code is instrumented with `TAILCHECK`. The cases were drawn from prior empirical studies [36, 65] that analyzed the types and frequencies of heap overflows in 85 CVEs. For example, our test suite includes the following cases:

- Loop accessing heap-allocated arrays, representing 35/85 studied CVE cases (41%).
- `memcpy()`, `memset()`, or `memmove()` into an insufficiently large buffer; 18/85 cases (21%).
- `strncpy()`, `strncmp()`, or `sprintf()` into an insufficiently large buffer; 6/85 cases (7%).
- Incorrect pointer arithmetic; 8/85 cases (9%).
- Accessing a derived class member on a base class object
- Attempting to iterate through `char*` cast to `long*`

Beyond the artificial test cases that we created, `TAILCHECK` also uncovered a heap overflow read in the SPEC CPU2017 `gcc` application (v4.5.0. function `vn_nary_may_trap` in `tree-ssa-sccvn.c:3365`). When instrumented with `TAILCHECK`, the application triggered a segmentation fault and produced a core file pointing to the error. This heap overflow was present in the code for 16 months before being detected with Valgrind (PR

	average latency	99th% latency	memory
apache	4% (3~6%)	3% (1~5%)	26% (15~32%)
nginx	2% (1~3%)	3% (1~5%)	41% (32~45%)
nginx (w/o poolalloc)	4% (3~6%)	3% (0~6%)	49% (44~52%)
memcached	3% (2~3%)	4% (3~5%)	2% (1~3%)
redis	6% (5~7%)	4% (0~18%)	3% (1~5%)
(Mean)	4%	3%	17%

Table 1: TAILCHECK runtime overhead (average latency and 99th% latency) and memory overhead on server applications, normalized to an uninstrumented base system. The latencies and memory overhead slightly vary for different file and object sizes tested. The first percentage is a geometric mean and the two numbers in parenthesis represent the range. The overall geometric (Mean) is computed for four servers, excluding nginx (w/o poolalloc), across all input sizes.

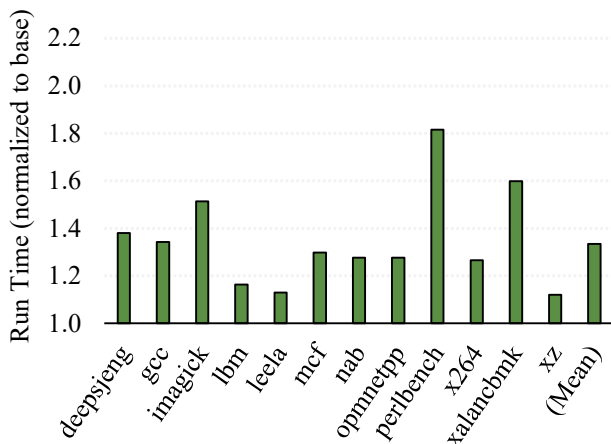


Figure 3: TAILCHECK run-time overhead on SPEC CPU2017, normalized to an uninstrumented base system.

tree-optimization/44124 [1]). The bug made it into the SPEC CPU2017 suite v1.0.5<sup>1</sup> and, to the best of our knowledge, TAILCHECK is the first to report it in the literature.

## 6.2 TAILCHECK Performance

We present the performance overhead of TAILCHECK on server applications in Table 1, which shows the average and 99th-percentile latencies. Both latencies vary only slightly for different test input sizes: 32KB, 128KB, 512KB, and 2MB files for web servers and 32B, 128B, 512B, and 2KB objects for key-value stores. There were no noticeable differences for nginx with and without application-level pool allocations. Redis with 2KB objects shows the highest 99th% latency overhead of 18%, yet with high variance. All the rest, in-

<sup>1</sup>The bug patch has been merged to gcc v4.5.1. The ChangeLog of SPEC CPU2017 does not indicate version update or bug fix.

cluding redis with smaller objects, show minor performance degradation ( $\leq 7\%$ ). An individual 99th-percentile latency result for different file/object sizes can be also found in Figure 5 (for the comparison with AddressSanitizer [54]). The geometric mean across the four servers was 4% and 3% for the average and 99th-percentile latencies, respectively.

The TAILCHECK performance results for SPEC CPU2017 are shown in Figure 3. The geometric mean of the TAILCHECK performance overhead for SPEC CPU2017 is 33%, among which perlbench shows the highest 1.8x slowdown. We present a performance comparison study with prior art in §6.5. Overall, we find that the combination of these servers and SPEC CPU performance results indicate overheads that are likely low enough to warrant production use of TAILCHECK for run-time heap overflow detection in security-conscious environments.

## 6.3 TAILCHECK Memory Usage

In addition to the performance overheads, TAILCHECK increases application memory requirements because it reserves space for the TailObjects at the end of each allocator managed region. Table 1 (last column) shows the memory overhead for the server applications. The relative increase in memory usage was small for the key-value store applications, while nginx shows the highest overheads. In TAILCHECK, a protected page for small objects is a virtual page with no access permission and thus does not require a physical page. However, for large objects, TAILCHECK still requires one TailObject and one protected page. Upon further investigation, we found that at start-up, nginx allocates a large number of large objects, incurring a relatively high memory overhead. However, we also observed that once initialized, its peak RSS does not change while serving client requests. Nginx (w/o poolalloc) allocates more (non-pool) large objects, showing a slightly higher memory overhead than nginx with pool allocations.

Next, we present the memory overheads in Table 2 for SPEC CPU2007 applications. Because TAILCHECK shares the space of the TailObjects for small objects within a region, the capacity overheads is minimal. The most affected benchmarks (perlbench, gcc, and nab) experience only a 9% increase in the peak RSS. The geometric means were 17% for the servers and 3% for the SPEC CPU2017 applications.

## 6.4 Analysis of Optimizations.

To better understand the TAILCHECK performance overheads, we analyze the benefits of the optimizations described in §3.2. For this experiment, we used the SPEC CPU2006 benchmark instead of CPU2017 because when we compare ours with Delta Pointers in §6.5, we want to compare the impact of the same static optimization (SafeAlloc) on ours and Delta Pointers. However, few benchmark applications in SPEC CPU2017 have memory requirements and cannot be supported by Delta

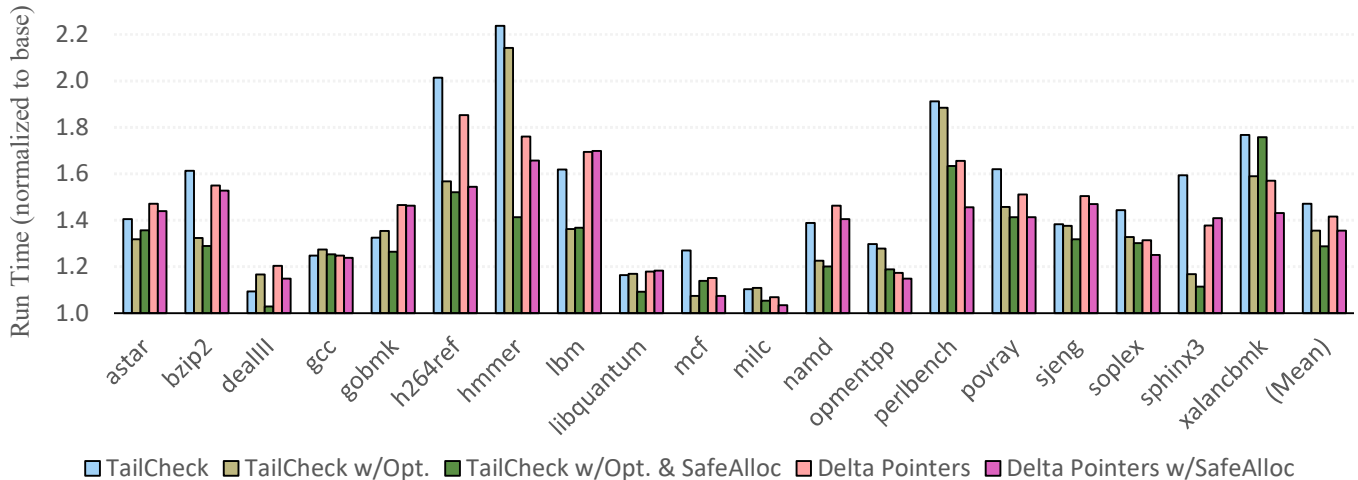


Figure 4: Comparison of run-time overheads on SPEC CPU2006, normalized to an uninstrumented base system: (a) TAILCHECK; (b) TAILCHECK with Opt.; (c) TAILCHECK with Opt. and SafeAlloc; (d) Delta Pointers; and (e) Delta Pointers with SafeAlloc.

application	overhead	application	overhead
deepsjeng	0%	nab	9%
gcc	9%	omentpp	3%
imagick	0%	perlbench	9%
lbm	0%	x264	6%
leela	-1%	xalancbmk	5%
mcf	0%	xz	0%
		geo-mean	3%

Table 2: TAILCHECK memory overhead (peak RSS) on SPEC CPU2017, normalized to an uninstrumented base system.

Pointers: *e.g.*, `xz` and `mcf` with the reference input. Thus, we based our analysis on SPEC CPU2006.

The first three bars in Figure 4 present the impact of optimizations. We first disable the merging of tag handling code when multiple offsets of an object are dereferenced within the same function. Although not drastic, the geometric mean of performance across the SPEC CPU2006 suite improves by 11%, with the biggest gains coming from several applications such as `bzip2`, `h264ref`, and `sphinx3`.

We also examine the benefits of applying SafeAlloc to avoid TAILCHECK instrumentation for heap objects whose accesses are known to be within bounds through static code analysis. Although the gains across all benchmarks are modest, 7% on average, applications such as `hmmer` and `perlbench` exhibit drastic benefits, reducing the run-time overheads by 73% and 24%, respectively. These applications have hot loops iterating over multiple large arrays, allowing SafeAlloc to find a significant number of optimization opportunities.

We note that “Hoisting Tag Handling,” as described in 3.2,

reduces address calculation overheads, but may also increase register pressure. For loops with a small number of pointer dereferences in the loop body, hoisting the computation of the tail pointer may add register pressure to the program, leading to performance degradation. The TAILCHECK compiler performs a simple count of the number of pointer dereferences, applying hoisting if a loop has two or more dereferences. In some cases (*e.g.*, `xalancbmk`), SafeAlloc eliminates some pointer dereferences, leaving just one dereference in the loop body, bypassing optimization in those loops.

## 6.5 Comparison with Delta Pointers

This experiment compares TAILCHECK with Delta Pointers, the state-of-the-art compiler-based solution that shares many similarities with TAILCHECK in that both use pointer tagging and do not perform explicit bound checking. We use `mimalloc`’s unmodified allocator for baseline and Delta Pointers performance measurements.

Figure 4 shows the performance comparison. First, when comparing the last two bars, we can find that the SafeAllocs optimization gives roughly the same relative benefit for Delta Pointers (6%) as for TAILCHECK (7%). One thing to note is that the other two tag merging and hoisting optimizations (excluding SafeAllocs) in §3.2 are only applicable to TAILCHECK, but not to Delta Pointers. The reason is that the two optimizations require the tag of a pointer remain unchanged once defined (until an object becomes freed), which is the case for TAILCHECK, but not for Delta Pointers.

Second, when comparing the two fully optimized versions, the 3rd and 5th bars in Figure 4, TAILCHECK exhibits lower runtime overhead than Delta Pointers: 29% vs. 35%. TAILCHECK has lower than or similar runtime overheads than Delta Pointers for most applications. Two exceptions were

`perlbench` and `xalancbmk`. There are two significant differences in Delta Pointers’ and TAILCHECK code instrumentation. Delta Pointers instruments pointer arithmetic to update a pointer tag, while TAILCHECK does not. TAILCHECK adds additional memory operation on a pointer dereference, while Delta Pointers does not. When considering the number of instrumentation as a factor of runtime overhead, TAILCHECK is likely to perform better than Delta Pointers for those applications with more pointer arithmetic and less dereferences.

For reference, we note that Oleksenko *et al.* [45] reported 1.8x, 1.8x, 2x, and >3x runtime overheads for AddressSanitizer [54], Intel’s MPX (ICC), SoftBound [40] and SAFE-Code [15], respectively, for the SPEC CPU2006 applications (in their experimental settings).

Delta Pointers does not incur additional memory overhead, as it does not use a custom allocator with guard pages like TAILCHECK (Table 2). Rather, the major drawback of Delta Pointers is the need to shrink the process address space (e.g., 32-bit tag and 32-bit address space).

## 6.6 Comparison with AddressSanitizer

Our last experiment compares TAILCHECK with AddressSanitizer [54] for server applications. AddressSanitizer is the state-of-the-practice solution that maintains a disjoint metadata space to distinguish safe regions and (unsafe) redzones. Figure 5 shows the 99th-percentile latency across different file sizes (32KB-2MB) for web servers and object sizes (32B-2KB) for key-value stores. As discussed in §6.2, TAILCHECK shows minor (on average 3%) tail latency degradation. The worst 18% overhead appears only for `redis` with 2KB objects. On the other hand, AddressSanitizer incurs higher overheads for all cases (on average 16%, up to 51%), reflecting its expensive metadata lookup and checking costs. Likewise, AddressSanitizer shows higher average latencies (not shown) than TAILCHECK: 4% vs. 12% on average; and 7% vs. 56% in the worst case.

## 7 Discussion

### 7.1 False Positives and False Negatives

TAILCHECK does not have false positives, assuming there are no use-after-free violations. A shadow memory access computed from a dangling pointer could be wrong if freed and reallocated objects have a different size. Otherwise, the tag in a pointer and the actual distance between a (current) object and its corresponding TailObject always match, and the size of a TailObject is always larger than or equal to that of an original object. Thus, any page fault from a protected page is evidence of a true heap overflow.

We exclude a discussion of potential segmentation faults from passing tagged pointers to uninstrumented code without

proper masking. The mechanisms for using tagged pointers in the presence of uninstrumented code are described in §4.

TAILCHECK may have false negatives (miss some heap overflows). First, TAILCHECK is a dynamic tool. It can detect a heap overflow only along the program paths that are explored at run-time, given a test input and environment. Second, TAILCHECK is an instrumentation-based tool and may miss a heap overflow in an object that crosses the instrumented vs. uninstrumented code boundary, such as calls into third-party libraries and assembly code.

Consider two cases, one in which a heap object is created in the instrumented code, but escapes unmasked into uninstrumented code where it is accessed, and vice versa. TAILCHECK cannot detect an overflow in uninstrumented code as there is no shadow TailObject access. Similarly, if an object allocated in the uninstrumented code is passed to instrumented code, TAILCHECK cannot detect an overflow as there is no tag and no corresponding protected page available for the object.

Finally, TAILCHECK relies on a guard page; thus it may fail to detect an overflow beyond the 4KB protected page (similar to AddressSanitizer’s 128B redzone [54]). However, it is difficult for a malicious user to exploit this, particularly for small objects, because both the original (manipulated) access and the TailCheck (shadow) access must land on legal memory regions to succeed. The TAILCHECK memory allocator scatters 64KB allocation regions for small objects in the process address space, making the distance between a protected page of a memory region and other valid memory regions non-deterministic. Large objects may be easier to exploit as their original and TailCheck accesses are to the same location. We note that this is different from AddressSanitizer’s traditional redzone approach in which a constant length (e.g., 128B) redzone is inserted between adjacent valid memory objects/regions. Prior solutions that use explicit bounds checking (e.g., MPX [45]) or precisely keep track of pointer arithmetic (e.g., Delta Pointers [28]) do not have this limitation.

### 7.2 Benign False Negatives due to Alignment

x86-64 Linux assumes that heap allocators return 16-byte aligned pointers, allowing the compiler to emit memory instructions based on this assumption. As discussed in §3.1, TAILCHECK enforces the same alignment for TailObjects as for the original objects, allowing the compiler to use the same memory instruction for both original and shadow memory accesses. For objects that are not 16-byte aligned (e.g., 11 byte), the bound of the corresponding 16-byte aligned TailObject (of the same 11 byte size) will not be adjacent to the boundary of a protected page and there will be a gap due to the alignment requirement (5 bytes in this example). This gap may lead to a false negative as the shadow access would not lead to a page fault. Nonetheless, we see this as a “benign” overflow, as the original object would also have the same gap before its adjacent object, due to the same alignment requirement.



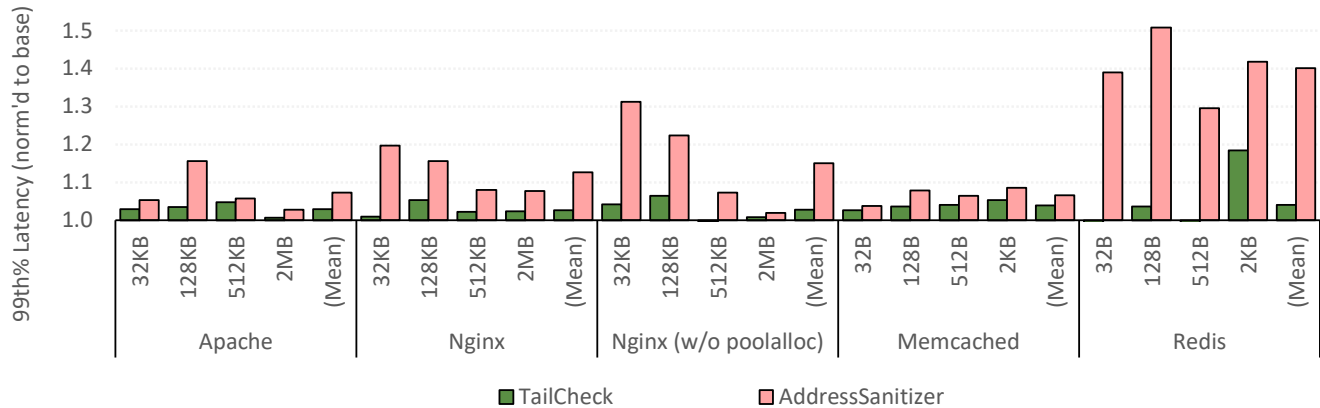


Figure 5: Comparison of 99th-percentile latency on server applications across different file/object sizes, normalized to an uninstrumented base system: (a) TAILCHECK and (b) AddressSanitizer.

If required, TAILCHECK can be extended to put the bound of a TailObject immediately before the protected page without a gap. TAILCHECK would create an unaligned TailObject, and there will not be a false negative due to alignment. However, in this case, the shadow memory access instrumentation pass may need to use different instruction opcodes for the shadow accesses, because the instructions used to access the original object may not support unaligned addresses. Although such a change is possible to eliminate these benign false positives, it is likely to come at a performance cost.

### 7.3 Potential Hardware Support

TAILCHECK performance could benefit from the following hardware support. First, TAILCHECK (on x86-64) must currently mask the tags of pointers before accessing an original object and before passing pointers to uninstrumented code. TAILCHECK could take advantage of the top-byte-ignore feature of ARM’s MTE [3] to avoid masking overhead, similar to HWAsan [55], a hardware-assisted ASan.

Second, TAILCHECK (on x86-64) relies on load and store instructions to perform shadow memory accesses for overflow detection. It would be sufficient for TAILCHECK shadow operations to only check for access permission. TAILCHECK could make use of new pseudo load/store-like instructions which perform virtual to physical address translation and check permissions, without performing an actual memory access or perturbing the data cache, eliminating cache pollution, cache coherence traffic, etc. Such shadow accesses would not modify memory, reducing the TAILCHECK run-time overhead and reducing the memory overhead because TailObject space would no longer need to be reserved.

Lastly, one can design a hardware TAILCHECK without compile-time dereference instrumentation. Given a tagged pointer, the memory management unit of a processor can transparently perform a page permission check or a shadow memory access to the TailObject.

### 7.4 Extensions to Other Memory Safety

**Heap Intra-Object Overflow.** TAILCHECK defines a heap object protection granularity at the time of heap allocation. Thus, TAILCHECK does not protect a more fine-grained sub-object from an overflow (e.g., an overflow of an array field of a struct to another field of the same struct) as in per-object bound checking solutions [2, 13, 17, 18, 25, 30, 51, 67]. If desired, TAILCHECK’s compiler instrumentation pass could be extended to “heapify” a subobject, similar to CCured [43]. This is analogous to the additional “bound narrowing” feature in some per-pointer bound checking solutions [40, 45].

**Heap Underflow.** Though buffer underflow is less critical than overflow in terms of security, if desired, the design of TAILCHECK could be flipped to “HeadCheck.”

**Heap Use-After-Free.** TAILCHECK does not support any temporal memory safety, yet it could be combined with existing use-after-free detection schemes that do not use pointer tagging: e.g., Oscar [12] or DangZero [20] that rely on page protection could be a good candidate for integration.

**Stack Overflow.** TAILCHECK assumes that stack is protected by other schemes such as stack canaries [11] and shadow stacks [60]. In the current form, TAILCHECK’s instrumentation pass does not need to distinguish stack and heap objects as any address-taken stack object would hold a tag of 0, leading to harmless redundant memory accesses.

If desired, TAILCHECK can be extended to support stack overflow protection. The simplest solution is to replace stack allocation with heap allocation, similar to CCured’s “heapified” stack [43], at some performance cost. Alternatively, TAILCHECK could be extended to add a protected page to stack and protect the stack objects similar to heap objects (using a distance tag, a TailObject, and a shadow memory access),

with the following instrumentation pass changes. The size of a TailObject (a max of projected objects) can be determined as the sizes of the stack objects are known. The location of TailObject (before a protected page) should be kept in a reserved register or a global variable by instrumenting the entry function: *e.g.*, `main`. For each function, any address-taken stack object (*e.g.*, defined by LLVM's `alloca`) should be instrumented to tag the distance from the stack object (whose address is computed from a stack pointer) to the TailObject (whose address is kept separately). Then, TAILCHECK can use the same mechanism for stack objects as heap objects. The default size of TAILCHECK's tag is 16 bits, implying that it can support a stack up to 64KB. If a larger stack is needed, the address space should be reduced for a wider tag. Selectively using heapification for a large stack object could be helpful.

## 8 Related Work

There are hundreds of prior memory safety solutions, with a little bit of exaggeration. This section does not attempt to cover them exhaustively. Instead, we focus on discussing where TAILCHECK sits among these related works.

### 8.1 Buffer Overflow Detection

The first group maintains “redzone” metadata and checks if a program accesses the red zone on each memory access. Purify [22] is the first to use redzone. LBC [21] introduces a fast path optimization skipping metadata lookup with a random canary. ASan [54] and Valgrind [44] are popular redzone-based tools using static instrumentation and dynamic binary translation, respectively.

The second group performs explicit “bounds checking.” Some maintain per-object bound metadata and perform bounds checking on pointer arithmetic: *e.g.*, J&K [25], CRED [51], D&A [13], Baggy Bounds [2], PAriCheck [67], LowFat [17, 30], and EffectiveSan [18]. Others keep track of per-pointer bound metadata and check bounds on pointer dereferences: *e.g.*, SoftBound [40], SGXBounds [29], Mid-Fat [27], MPX [45], CUP [8], and FRAMER [42]. Static analysis can be used to avoid some bound checks on memory accesses proven to be safe: *e.g.*, PICO [26]. The pointer-based approach has another advantage that makes it easy to support intra-object overflow protection: *e.g.*, an array in a struct.

The third group leverages “page protection”: *e.g.*, Electric Fence [49], DUMA [5], DYBOC [56], libgmalloc [35], and PageHeap [61]. They do not maintain redzone/bound metadata nor perform explicit checking as in the above two groups. However, as discussed in §2.3, allocating one object per page incurs huge memory and run-time overheads. Prober [37] shows low overhead but it only protects heap arrays. TAILCHECK proposes a new low-overhead page protection-based solution for all heap objects.

On the other hand, Delta Pointers [28] check the integer overflow of a tagged pointer. It does not make use of a redzone, a bound, or a protected page; and thus does not fall into any of the above groups.

### 8.2 Pointer Tagging

Many of the above solutions need to maintain some metadata. Some use “fat pointers” that stores the metadata (*e.g.*, base and bound) in separate words alongside the actual pointer value. Examples include Safe-C [4], Cyclone [24], and CCured [43]. CHERI [62, 63] provide hardware support for fat pointers.

Many recent works leverage “pointer tagging” that embeds metadata into some bits of a pointer itself, to avoid a code layout change. For example, Baggy Bounds [2] uses the spare top bits to store the distance between an out of bound pointer and its intended referent. Delta Pointers [28] uses a 32-bit tag to encode the distance from the current pointer to the end of an object. As discussed in §2.3, one common downside of pointer tagging is that it may restrict the address space: *e.g.*, Delta Pointers only support a 4GB of 32-bit address space. This may not be a problem for SGXBounds [29], which is designed for an Intel SGX enclave with already-limited 32-bit address space, and thus it can use a 32-bit tag to store the upper bound of the pointer's referent without any sacrifice. However, other pointer tagging solutions (including Delta Pointers) that require more than 16 unused bits in the current 64 bit architecture cannot be used for general (non-SGX) programs with big memory requirements. TAILCHECK does not share this limitation. Alternatively, CUP [8] takes an extreme design that uses the entire pointer width to store tags. Low-fat pointers [17, 30] store the tag implicitly in the pointer value, and thus can be safely dereferenced without masking.

Several works proposed hardware support for pointer tagging. In-Fat [64] is a hardware extension of EffectiveSan [18], which tags the upper bits of a pointer with an index into a bounds table, performing bounds checking on pointer arithmetic. HeapCheck [52] stores an index into a per-pointer bounds table, and checks bounds on pointer dereferences. PACMem [34] leverages ARMv8 AArch64 Pointer Authentication (PA) [50], computing cryptographic hashes based on the value of pointers (and other contexts) for pointer integrity. PACMem seals object metadata into the high-order bits of pointers via PA and uses the seal as the index to retrieve it. The tagged PA codes are propagated by hardware along with the pointers. No-Fat [23] supports low-fat pointers [17, 30].

### 8.3 Use-After-Free Detection

Existing use-after-free solutions can be categorized into three groups based on their detection techniques. Some solutions such as CETS [41], ViK [10] and xTag [6] tag the allocated memory and the pointer with a unique identifier (referred to as lock and key), and check if the tags of pointer and memory

match on dereference. Any mismatch indicates that a pointer used for dereference is a dangling pointer. ARM's Memory Tagging Extension (MTE) [3] and SPARCS's Silicon Secured Memory (SSM) [46] provide hardware support to assign random 4-bit tags to object-pointer pairs to probabilistically find use-after-free bugs on tag mismatch. HWAsan [55] an extension of ASan with ARM's MTE makes use of its top-bit-ignore feature and avoids masking on memory dereference.

Other solutions such as Undangle [9], FreeSentry [66], DangleNull [32], DangSan [59], BOGO [68] maintain metadata to find and invalidate dangling pointers on free. Then a use-after-free is detected as an invalid pointer use: *e.g.*, null pointer dereference.

Yet others such as D&A [14], Oscar [12], and DangZero [20] leverage page protection: a page becomes inaccessible after a free. Oscar [12] reduces physical memory and run-time overhead by mapping multiple virtual pages into a single physical page. DangZero [20] further lowers run-time overhead by directly accessing the page tables with support from virtualization extensions and a privileged backend (*e.g.*, Kernel Mode Linux). TAILCHECK does not provide use-after-free detection, but its page-based approach makes it possible to integrate the above page-based use-after-free solutions to achieve both spatial and temporal memory safety. We leave this to future work.

## 8.4 Uninitialized Memory Read

Uninitialized memory reads can lead to information leakage, similar to buffer overflow reads. Purify [22] and Valgrind [44] detect an uninitialized memory read by maintaining and checking (initialized vs. uninitialized) state metadata at a byte or bit granularity, respectively. UniSan [38] uses data-flow analysis to zero-out variables that might be disclosed to an attacker. SafeInit [39] modifies the compiler and heap allocator to ensure that all stack/heap regions be initialized.

## 9 Conclusions

Heap overflow vulnerabilities leave many software systems exposed to security attacks and exploitation. This work presented TAILCHECK, a novel heap overflow mitigation scheme that leverages a custom memory allocator, OS-based page protection, and compiler-directed pointer tagging. TAILCHECK achieves low run-time overhead by detecting heap overflows using page protection, without maintaining bound metadata and without performing explicit bounds checks. TAILCHECK uses pointer tagging and shadow memory accesses to detect overflows, allowing multiple original objects to share a single TailObject, which reduces both performance and memory overheads compared to the previously explored techniques. The results of our experimental evaluation demonstrate the effectiveness and efficiency of TAILCHECK in detecting heap overflows in C and C++ programs.

## Acknowledgements

We would like to thank the anonymous reviewers for their feedback on the draft of this work, and James Mickens for acting as our shepherd. This work was supported in part by the National Science Foundation under Grant No. CCF-2153747, CNS-2135157, CNS-2214980, CCF-2153297, and CNS-1763680.

## References

- [1] re PR tree-optimization/44124. <https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=4d085b9ee391f005d209512de3ea283fde49d42e>.
- [2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, 2009.
- [3] ARM. Armv8.5-a memory tagging extension. [https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm\\_Memory\\_Tagging\\_Extension\\_Whitepaper.pdf](https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf).
- [4] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.
- [5] Hayati Aygün. Detect Unintended Memory Access (D.U.M.A.). <https://duma.sourceforge.io/>, 2022. [Online; accessed 29-Nov-2022].
- [6] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. xtag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on intel x86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 502–519, 2022.
- [7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.
- [8] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 381–392, New York, NY, USA, 2018. Association for Computing Machinery.

- [9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, page 133–143, New York, NY, USA, 2012. Association for Computing Machinery.
- [10] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Vik: Practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, page 271–284, New York, NY, USA, 2022. Association for Computing Machinery.
- [11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.
- [12] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.
- [13] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171, 2006.
- [14] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280. IEEE, 2006.
- [15] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, page 144–157, New York, NY, USA, 2006. Association for Computing Machinery.
- [16] Jaana Dogan. hey: Http load generator. <https://github.com/rakyll/hey>.
- [17] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016*, page 132–142, New York, NY, USA, 2016. Association for Computing Machinery.
- [18] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery.
- [19] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference, IMC '14*, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.
- [20] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, page 1307–1322, New York, NY, USA, 2022. Association for Computing Machinery.
- [21] Niranjana Hasabnis, Ashish Misra, and R Sekar. Lightweight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.
- [22] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*, pages 125–136, 1992.
- [23] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 916–929, 2021.
- [24] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
- [25] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUD*, volume 97, pages 13–26, 1997.
- [26] Tina Jung, Fabian Ritter, and Sebastian Hack. Pico: A presburger in-bounds check optimization for compiler-based memory safety instrumentations. 18(4), jul 2021.
- [27] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.



- [28] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.
- [29] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.
- [30] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732, 2013.
- [31] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [32] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.
- [33] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*, pages 244–265. Springer, 2019.
- [34] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, page 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery.
- [35] libgmalloc(3) [osx man page]. (guard malloc), an aggressive debugging malloc library. <https://www.unix.com/man-page/osx/3/libgmalloc/>.
- [36] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. Prober: Practically defending overflows with page protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20*, page 1116–1128, New York, NY, USA, 2021. Association for Computing Machinery.
- [37] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. Prober: Practically defending overflows with page protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20*, page 1116–1128, New York, NY, USA, 2021. Association for Computing Machinery.
- [38] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 920–932, New York, NY, USA, 2016. Association for Computing Machinery.
- [39] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, volume 17, pages 1–15, 2017.
- [40] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.
- [41] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management, ISMM '10*, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.
- [42] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 612–626, New York, NY, USA, 2019. Association for Computing Machinery.
- [43] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.
- [44] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [45] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.
- [46] Oracle. Sparc m7 silicon secured memory (ssm). [https://docs.oracle.com/cd/E37069\\_01/html/E37085/gphwb.html](https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html).

- [47] Mitre Org. CVE Buffer Overflow. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow>, 2022. [Online; accessed 29-Nov-2022].
- [48] Mitre Org. CWE-787. <https://cwe.mitre.org/data/definitions/787.html>, 2022. [Online; accessed 29-Nov-2022].
- [49] Bruce Perens. Electric Fence. <https://linux.die.net/man/3/efence/>, 1987. [Online; accessed 19-Nov-2022].
- [50] Qualcomm. Pointer authentication on armv8.3. <https://www.qualcomm.com/media/documents/files/whitepaper-pointerauthentication-on-armv8-3.pdf>, 2017.
- [51] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 4, pages 159–169, 2004.
- [52] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: Low-cost hardware support for memory safety. *ACM Trans. Archit. Code Optim.*, 19(1), jan 2022.
- [53] R.C. Seacord. *Secure Coding in C and C++*. SEI series in software engineering. Addison-Wesley, 2013.
- [54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.
- [55] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyurkevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety, 2018.
- [56] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th International Conference on Information Security, ISC’05*, page 1–15, Berlin, Heidelberg, 2005. Springer-Verlag.
- [57] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.
- [58] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.
- [59] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsán: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys ’17*, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery.
- [60] Vencidator. Stack Shield: A “stack smashing” technique protection tool for linux. <https://www.angelfire.com/sk/stackshield/>, 2000.
- [61] Microsoft Windows. PageHeap. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap>, 2022. [Online; accessed 21-Nov-2022].
- [62] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.
- [63] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’52*, page 545–557, New York, NY, USA, 2019. Association for Computing Machinery.
- [64] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21*, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.
- [65] Tao Ye, Lingming Zhang, Linzhang Wang, and Xuan-dong Li. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 91–101, 2016.
- [66] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.
- [67] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. Parichck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, 2010.

- [68] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 631–644, New York, NY, USA, 2019. Association for Computing Machinery.



# SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory

Xuchuan Luo<sup>1,\*</sup>, Pengfei Zuo<sup>2</sup>, Jiacheng Shen<sup>3,\*</sup>, Jiazhen Gu<sup>3</sup>,  
Xin Wang<sup>1,4</sup>, Michael R. Lyu<sup>3</sup>, and Yangfan Zhou<sup>1,4</sup>

<sup>1</sup>*School of Computer Science, Fudan University*

<sup>2</sup>*Huawei Cloud*      <sup>3</sup>*The Chinese University of Hong Kong*

<sup>4</sup>*Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*

## Abstract

Disaggregated memory (DM) is an increasingly prevalent architecture in academia and industry with high resource utilization. It separates computing and memory resources into two pools and interconnects them with fast networks. Existing range indexes on DM are based on B+ trees, which suffer from large inherent read and write amplifications. The read and write amplifications rapidly saturate the network bandwidth, resulting in low request throughput and high access latency of B+ trees on DM.

In this paper, we propose to use the radix tree, which is more suitable for DM than the B+ tree due to smaller read and write amplifications. However, constructing a radix tree on DM is challenging due to the costly lock-based concurrency control, the bounded memory-side IOPS, and the complicated computing-side cache validation. To address these challenges, we design **SMART**, the first radix tree for disaggregated memory with high performance. Specifically, we leverage 1) a *hybrid concurrency control* scheme including lock-free internal nodes and fine-grained lock-based leaf nodes to reduce lock overhead, 2) a computing-side *read-delegation and write-combining* technique to break through the IOPS upper bound by reducing redundant I/Os, and 3) a simple yet effective *reverse check* mechanism for computing-side cache validation. Experimental results show that SMART achieves  $6.1\times$  higher throughput under typical write-intensive workloads and  $2.8\times$  higher throughput under read-only workloads, compared with state-of-the-art B+ trees on DM.

## 1 Introduction

Distributed range indexes are fundamental building blocks of many applications, *e.g.*, databases and key-value stores, to conduct range queries [2, 21, 53, 57, 59]. To improve resource utilization, many new proposals adopt the disaggregated memory (DM) architecture [53, 59]. DM can decouple computing and memory resources into two elastic resource pools (*i.e.*,

computing pool and memory pool) interconnected with high-speed networks, *e.g.*, remote direct memory access (RDMA) connections [3, 9, 16, 19, 20, 27, 47]. In this way, a DM range indexing system can utilize resources more efficiently.

Current DM index systems [53, 59] use B+ tree to build range indexes, following the idea generally adopted in the monolithic server solutions. However, B+ trees can bring severe read and write amplification issues on DM. Specifically, when reading or writing a key-value item in a B+ tree, one should search the tree by traversing many nodes which contain many useless keys and pointers since only one key is the target. This inevitably amplifies the network bandwidth consumption. As such network bandwidth is generally the bottleneck of the DM architecture [23], the amplified bandwidth consumption incurred by B+ trees exacerbates the bottleneck. This issue will lead to low overall throughput and high access latency. Our experimental study shows that it can dramatically degrade the throughput of Sherman [53], the state-of-the-art B+ tree index on DM. The throughput is  $10.8\times$  lower than the theoretical bound of RNICs under the YCSB workloads [10].

In this paper, we propose that radix tree is a more suitable tree index structure for DM. Compared with B+ trees, radix trees have smaller read and write amplifications since they do not store the entire keys in internal nodes. Moreover, the state-of-the-art radix tree design, *i.e.*, ART [32], further reduces read and write amplifications with an adaptive internal node design. However, several challenges should be addressed before radix trees become a high-performance, practical indexing solution for DM.

**(1) Lock-based concurrency control is expensive.** Remote lock operations are expensive on DM. However, the existing ART design adopts a lock-based algorithm for concurrency control [33], which contains many remote lock operations, worsening the write performance. In addition, computing-side caches are required on DM to reduce operation latency. The traditional read-copy-update (RCU) scheme for radix trees causes frequent changes in the addresses of cached nodes, leading to cache thrashing.

**(2) Redundant I/Os deteriorate the throughput.** RNICs

\*The work was mainly conducted when Xuchuan and Jiacheng were interns at Huawei Cloud.



in the memory pool of DM have bounded IOPS (I/O per second) [51]. However, radix trees have multiple small-sized read and write operations when traversing and modifying the tree index. Many of these read and write operations are redundant when multiple clients on the same compute node concurrently traverse the tree. These redundant I/Os on DM waste the limited IOPS of RNICs and thus decrease the peak throughput of radix trees.

(3) *The complicated computing-side cache validation.* Tree indexes on DM typically adopt computing-side caches to reduce access latency [56]. However, the structural features of radix trees (e.g., path compression) incur many address changes and metadata changes in radix tree nodes. These changes add more cache invalidation situations and thus complicate the cache design.

To address the above challenges, we propose **SMART**, a **dis**aggregated-me**M**emory-friendly **A**daptive **R**adix **T**ree. First, for better concurrency control, we present a *hybrid ART concurrency control* scheme with a lock-free internal node design and a lock-based leaf node design to achieve high performance without cache thrashing. Second, for an IOPS breakthrough, we propose a *read-delegation and write-combining (RDWC)* technique to reduce computing-side redundant I/Os. Third, for cache validation, we co-design SMART with an *ART cache*, including a reverse check mechanism to handle new cache invalidation situations of ART.

We implement SMART from scratch and evaluate it using the YCSB benchmark [10]. Compared with Sherman [53], the state-of-the-art B+-tree-based range index on DM, SMART achieves up to  $6.1\times$  higher throughput and  $1.4\times$  lower latency for typical write-intensive workloads and  $2.8\times$  higher throughput with similar latency for read-only workloads. The code of SMART is available at <https://github.com/dme/msys/SMART>.

In summary, this paper makes the following contributions:

- We propose that ART is a better tree index on DM, based on theoretical analysis and experimental results.
- We present the first memory-disaggregated radix tree, SMART, with three key designs for high performance, including a hybrid ART concurrency control scheme, a read-delegation and write-combining technique, and a reverse check mechanism for cache validation.
- We implement SMART and evaluate it using YCSB workloads [10]. The evaluation results demonstrate the efficacy and efficiency of SMART.

## 2 Background

### 2.1 Disaggregated Memory Architecture

As shown in Figure 1, the DM architecture physically separates computing (e.g., CPUs) and memory (e.g., DRAM) resources into two independent resource pools to address the resource utilization issue in traditional data centers with monolithic servers [18, 31, 42, 43, 46, 54]. In the DM architecture, compute nodes (CNs) own powerful computing resources but

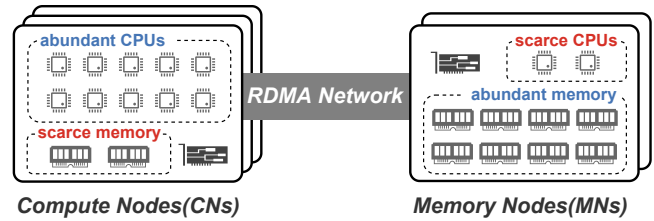


Figure 1: The architecture of disaggregated memory.

only have a small piece of memory serving as local caches. In contrast, memory nodes (MNs) are equipped with masses of memory but only own a few wimpy computing cores for simple tasks such as establishing network connections and allocating memory spaces.

A high-speed network with high bandwidth and low latency, e.g., RDMA network, is a crucial component in the DM architecture that interconnects CNs and MNs [12, 17]. RDMA network interface cards (RNICs) allow CNs and MNs to communicate with each other using *one-side verbs* (e.g., RDMA\_READ, RDMA\_WRITE, RDMA\_CAS) or *two-side verbs* (e.g., RDMA\_SEND, RDMA\_RECV). One-side verbs are preferred on the DM architecture to enable computing-side clients to operate directly on the disaggregated memory without involving the weak CPUs on MNs.

### 2.2 B+ Trees on Disaggregated Memory

Tree indexes are critical for many applications requiring range queries. All previously proposed tree indexes on DM are variants of the B+ tree, including FG [59] and Sherman [53]. FG is the first RDMA-based index supporting DM. It uses a B-link tree structure and completely leverages one-sided verbs to perform index operations, with RDMA-based spin locks for concurrency control. Since FG directly ports the spin-lock-based concurrency control and B-link tree node designs on monolithic servers to DM, its performance suffers from severe network contention on lock retries and write amplification on B-link tree nodes. Sherman [53] is the state-of-the-art B+ tree on DM that addresses the network contention and write amplification issues of FG. First, it addresses the network contention on lock-fail retries with a *hierarchical on-chip lock (HOCL)* scheme. The network requests on lock-fail retries are reduced with a local lock table shared among clients on the same CN. The on-chip memory of RNICs is leveraged to reduce PCIe transmissions further. Second, it mitigates the write amplification by allowing fine-grained modification to B+ tree nodes with a *two-level version mechanism*. Therefore, Sherman achieves much better performance than FG. However, Sherman still suffers from the natural performance bottleneck of B+ trees, i.e., coarse-grained lock-based concurrency control and inherent read amplification, which are analyzed in Section 3.

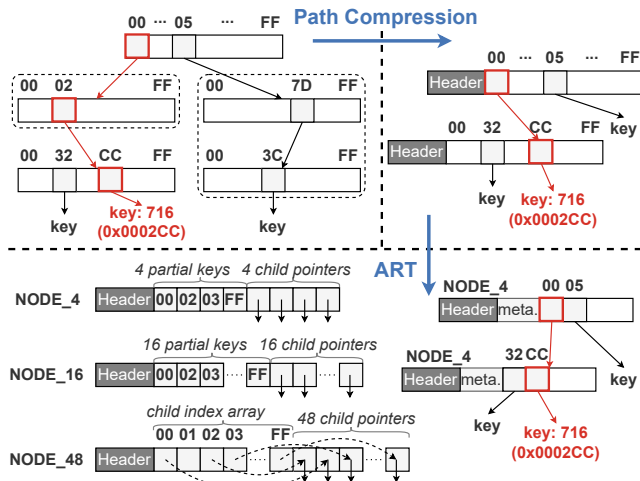


Figure 2: The optimization process from the basic radix tree to ART. For clarity, hexadecimal partial keys are shown. NODE\_256 is simply an array of 256 pointers, which is not shown due to space limitation.

## 2.3 Radix Tree

The radix tree is another popular tree index structure. It stores the segmented key in the top-down search path over the tree rather than storing the whole key in the internal node. Specifically, each internal node in the radix tree consists of an array of child pointers. Each pointer is associated with a segment of bits of the whole key, called *partial key*, as shown in Figure 2.

**Path compression.** Path compression is an optimization method for the radix tree to reduce tree height by removing one-child internal nodes, and can be implemented in three ways [32]: 1) *The optimistic method* simply abandons the partial keys in the removed nodes and instead stores a depth value to ensure the subsequent traversal process. 2) *The pessimistic method* stores all the partial keys of the removed nodes in the header of the subsequent node. 3) *The hybrid method* integrates the two methods above by storing partial keys into the fixed-sized header of the subsequent node, together with a depth value to ensure the subsequent traversal if some partial keys overflow from the header.

**Adaptive radix tree (ART).** ART [32] is the state-of-the-art variant of the 8-bit-span radix tree, designed to optimize the memory utilization of traditional radix trees. Traditionally, an internal node of a radix tree has all 256 pointers representing all possible partial keys. Many pointers are empty due to the sparse key distribution [32], wasting memory space in these internal nodes. ART addresses the issue by proposing four well-designed internal node structures with different numbers of pointers, *i.e.*, 4, 16, 48, and 256. It dynamically chooses the best-fit internal node structure to save memory space. As for concurrency control, ART is synchronized using a lock-based algorithm, *i.e.*, the *read-optimized write exclusion (ROWEX)* protocol [33]. There are some proposed ART-based indexes designed on monolithic servers [26, 29, 30, 37], while none of them is designed for DM.

Table 1: Read and write amplification factors of different trees.

	ART	B+ Tree	Sherman
Read	$\frac{M_1+E}{E} = 1.10$	$\frac{M_2+S \cdot E}{E} = 32.7$	$\frac{M_2+S \cdot (M_3+E)}{E} = 33.0$
Write	$\frac{M_1+E}{E} = 1.10$	$\frac{M_2+S \cdot E}{E} = 32.7$	$\frac{M_3+E}{E} = 1.01$

## 3 Analysis of Tree Indexes Built on DM

In this section, we first theoretically and experimentally compare B+ trees with a vanilla ART (§ 3.1). We then present the challenges of designing ART on DM (§ 3.2).

All the experiments in this section are conducted with 8 CNs and 1 MN, each equipped with a 100Gbps Mellanox ConnectX-6 RNIC. Each CN launches 32 clients with one shared 600MB cache. We use YCSB workloads [10] (including 60 million entries) with 32-byte string keys and 64-byte values, which is typical in real-world workloads [4, 58].

### 3.1 Motivations: B+ Tree vs. ART on DM

The main problem of B+ trees on DM is their severe read and write amplifications. In internal nodes, the B+ tree stores the whole keys. In leaf nodes, the B+ tree stores multiple keys together. Without optimizations, the B+ tree needs to read and write the entire nodes during each index operation, causing serious read and write amplifications. In the following, we first theoretically compare the read and write amplifications of ART with the B+ tree and the write-optimized B+ tree (*i.e.*, Sherman [53]). We then experimentally show the performance impacts due to the read amplification.

#### 3.1.1 Theoretical Analysis

The read and write amplification factors of different tree structures are shown in Table 1, respectively. We assume the internal nodes are cached and no node split occurs for brevity.  $M_1$  and  $M_2$  denote the metadata size of the leaf node of the radix tree and B+ tree, respectively.  $M_3$  denotes the size of the additional metadata (*i.e.*, entry-level versions) that Sherman applied to each key-value item.  $S$  denotes the span size of the B+ tree node.  $E$  denotes the key-value item size.

The amplification factor is defined as the ratio of bandwidth consumption from the server and bandwidth returned to the application. Without optimizations, when a client reads or writes a single key-value item in a tree index, the whole leaf node should be read or written. We use the same size of the key-value item, *i.e.*, 96 bytes, for all trees as an example.

The leaf node of the ART contains one item with its metadata. In our implementation, 10 bytes of metadata is enough for each item in ART. The read and write amplification factors are  $\frac{M_1+E}{E} = \frac{10B+96B}{96B} = 1.10$ .

The leaf node of the B+ tree contains  $S$  items together with the metadata. The metadata at least includes two fence keys ( $2 \cdot 32B$ ), a valid bit, a lock bit, a 1-byte level field, and two 7-bit versions [53], *i.e.*, 67 bytes in total. We use the

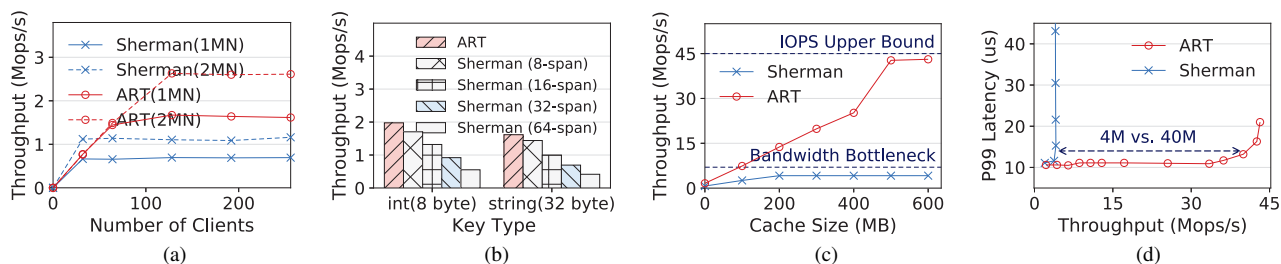


Figure 3: The read performances of Sherman and ART under the YCSB C workload (100% read). (a) The throughput bottleneck with no cache. (b) The impact of key size and span size with no cache. (c) The peak throughput with various sizes of caches. (d) The latency deterioration with excess requests.

default span size in Sherman, which is 32. The read and write amplification factors are  $\frac{M_2+S+E}{E} = \frac{67B+32\cdot 96B}{96B} = 32.7$ .

For Sherman, each key-value item in the leaf node is surrounded by a pair of 4-bit entry-level versions. Thus the read amplification factor is  $\frac{M_2+S\cdot(M_3+E)}{E} = \frac{67B+32\cdot(1B+96B)}{96B} = 33.0$ . When writing an item without node splitting, the client only requires to write back the modified item with its associated entry-level versions. Thus the write amplification factor is  $\frac{M_3+E}{E} = \frac{1B+96B}{96B} = 1.01$ .

### 3.1.2 Experimental Results

To show the impact of read amplification on the performance, we compare the performances of Sherman and ART under *read-only workloads*. The impact of write amplification is similar. We observe that the amplification leads to low throughput and high latency of B+ trees on DM.

**Observation 1: The throughput of the B+ tree is bounded by network bandwidth.** The memory-side network bandwidth is generally the performance bottleneck in the DM architecture [23]. The read and write amplifications of B+ trees cause more bandwidth consumption for each request, exacerbating the network bottleneck and resulting in low throughput.

As shown in Figure 3a, with an increasing number of clients, the limited bandwidth prevents the throughput of Sherman and ART from continually rising. With the same RNIC bandwidth, Sherman has a lower peak throughput than ART due to the severe read amplification. As shown in Figure 3b, the larger the key size or the span size (*i.e.*, the number of keys stored in a leaf node) is, the larger the read amplification is, which decreases the peak throughput of Sherman.

A computing-side cache is usually used for caching the internal nodes of the B+ tree on DM. As shown in Figure 3c, with the increasing size of the cache, the throughput of Sherman keeps bounded by the bandwidth bottleneck and finally saturates at 4.17 Mops/s. The bandwidth consumption from the server equals the maximum network bandwidth of 100 Gbps (12.5 GBps), and the bandwidth returned to the application is  $4.17 \text{ Mops/s} \cdot 96B = 0.39 \text{ GBps}$ . Thus the measured read amplification factor of Sherman is  $12.5 \text{ GBps} / 0.39 \text{ GBps} = 32.1$ , which is close to our theo-

retical analysis in § 3.1.1.

In contrast, without the read amplification from leaf nodes, the throughput of ART reaches about 45 Mops/s, which is the IOPS upper bound of the RNIC we use. This indicates that ART can make full use of the RNIC capacity and achieve the best resource efficiency as DM desires.

**Observation 2: The latency of the B+ tree is worsened by early network congestion.** Network congestion occurs when computing-side requests saturate the bandwidth or IOPS upper bound of RNICs. As the number of clients keeps growing, excess client requests need to queue up across the network, which results in latency deterioration. The read and write amplifications make B+ trees consume the bandwidth rapidly, expediting the process of network congestion.

As shown in Figure 3d, with the increase of throughput, the latency of Sherman and ART is stable in the beginning and then experiences a sudden surge due to the network congestion. Moreover, with the same memory-side RNIC bandwidth, Sherman has a much smaller inflection point (*i.e.*, the throughput threshold that triggers network congestion) than ART. As a result, Sherman shows an extremely high latency with relatively few clients. By contrast, ART has a high tolerance to this latency deterioration thanks to its small amplifications.

## 3.2 Challenges: ART on DM

Even though ART has superiority under *read-only workloads*, it suffers from significant challenges on DM under *hybrid read-write workloads*.

**Challenge 1: Lock-based concurrency control of ART causes poor write performance.** Existing ART adopts lock-based algorithms to perform synchronization [33]. However, lock operations are expensive on DM and lead to poor write performance, as shown in Figure 4a. Specifically, unlike local memory, each lock operation on DM requires additional network transmission (*e.g.*, RDMA\_CAS). Furthermore, the lock conflict mechanism (*i.e.*, busy waiting) causes frequent RDMA retries when failing to acquire a lock, which wastes the limited IOPS of RNICs and reduces the throughput.

One feasible solution is to design lock-free algorithms. However, lock-free design is not the best choice for ART as

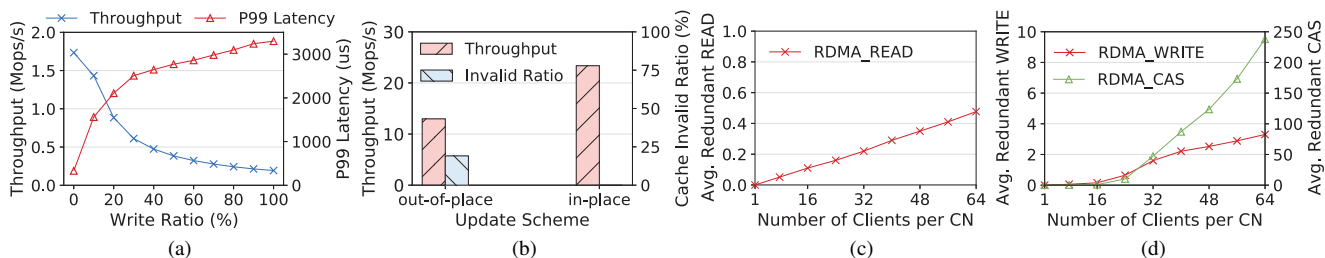


Figure 4: (a) The write performance of ART under the YCSB insert workload (100% insert) with no cache. (b) The performance degradation caused by cache thrashing under the YCSB A workload (50% read + 50% update) with sufficient caches. (c-d) The inter-client redundant I/Os on DM in terms of reads and writes.

well. Specifically, an out-of-place update scheme is required for lock-free algorithms to update items larger than 8 bytes. It atomically compares and swaps the corresponding 8-byte addresses instead of modifying the items in place, as the latter cannot be realized atomically. However, in high-concurrency scenarios, a mass of out-of-place updates lead to frequent changes in the addresses of items. This brings about the severe cache coherence issue since the old addresses of the items have been cached in other CNs. Even worse, in skewed workloads, the addresses of hot items are changed continuously and repeatedly, resulting in cache thrashing.

To verify this, we evaluate the two update schemes in ART with the YCSB A workload,<sup>1</sup> as shown in Figure 4b. The out-of-place scheme brings about an average of 19.1% invalid cached addresses of leaf nodes and thus results in a 44.5% throughput decline compared with the in-place scheme.

**Challenge 2: Inter-client redundant I/Os on DM waste the limited IOPS of RNICs.** As mentioned in **Observation 1**, B+ trees suffer from bandwidth bottleneck, while ART can break through the bottleneck and achieve the IOPS upper bound of RNICs, with small read and write amplifications.

However, we find that there are redundant I/Os that waste the limited IOPS of RNICs in the DM architecture, hindering ART from continually breaking through the IOPS upper bound. Specifically, taking read operations as an example, when several clients on the same CN read the same key-value item concurrently, they send identical RDMA\_READs across the network. This is superfluous duplication of effort since all these requests do the same transmission work.

To measure the extent of underlying inter-client redundant reads, we launch various numbers of clients on the same CN. Each client continuously issues 1KB RDMA\_READs, with their destination addresses following a Zipfian distribution of skewness 0.99 (*i.e.*, the same as YCSB’s). As shown in Figure 4c, during each read time window, the average number of redundant RDMA\_READs increases with the number of clients and achieves up to 0.48 with 64 clients, implying 48% read performance improvement potential.

<sup>1</sup>To eliminate the impact of concurrency conflicts, we scatter the update part of workloads among clients without intersection.

As for inter-client redundant writes, we issue constant RDMA\_WRITES with lock-based concurrency control via RDMA\_CASes from each client. As shown in Figure 4d, during each write time window (including lock acquisition and release), the average number of redundant RDMA\_WRITES grows and reaches up to 3.3, indicating around 330% write performance improvement space with 64 clients. Interestingly, the number of redundant writes is more than the read one since redundant writes inevitably exacerbate the concurrency conflicts, leading to a longer write time window and thus more redundant writes in return. The nearly exponential growth of the redundant number of RDMA\_CASes saturates the IOPS upper bound rapidly and causes poor write performance.

**Challenge 3: Structural features of ART deteriorate the problem of computing-side cache invalidation.** As presented in § 2.3, *path compression* and *adaptive nodes* are two important structural features that reduce memory consumption by reducing the tree height and the node size, respectively. However, these two features introduce new cache validation problems. For instance, adjustments on the parent-child relationship of nodes may happen during insertion into compressed nodes. The caches on other CNs still store the old content of the parent node. If a client on those CNs does not conduct a cache verification, it incorrectly reads the old child node according to the outdated cache and thus fails to access the newly inserted node. Similarly, node type changes are invisible by the computing-side cache either, which may lead to incomplete node fetching.

## 4 SMART Design

We propose SMART, a high-performance ART for DM. Figure 5 shows the overview of SMART. To improve the efficiency of concurrency control (**Challenge 1**), we present a *hybrid ART concurrency control* scheme. The scheme contains a lock-free internal node design and a lock-based leaf node design to achieve high write performance without cache thrashing (§ 4.1). To save the limited IOPS of RNICs (**Challenge 2**), we propose a *read-delegation and write-combining (RDWC)* technique to eliminate inter-client redundant I/Os (§ 4.2). To handle the cache validation (**Challenge 3**), we co-design SMART with an *ART cache* (§ 4.3), including a reverse check



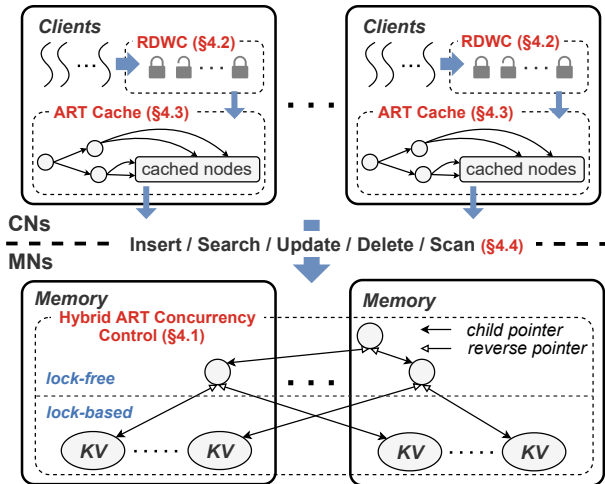


Figure 5: The overview of SMART.

mechanism. Lastly, we summarize the operations (*i.e.*, insert, search, update, delete, scan) that SMART supports (§ 4.4).

## 4.1 Hybrid ART Concurrency Control

In this section, we first describe the data structures and concurrent operations of the hybrid concurrency control scheme in SMART. We then introduce RDMA-related optimizations.

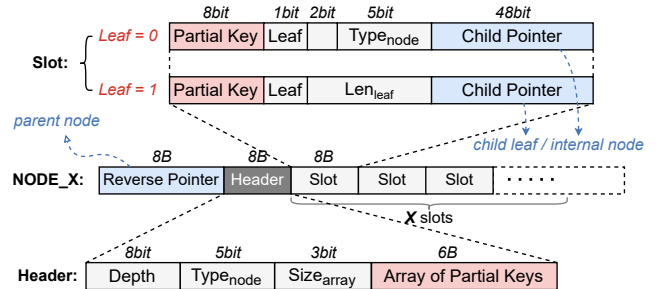
### 4.1.1 Data Structures

**Lock-free internal node.** As the addresses of internal nodes change more infrequently, internal nodes do not cause cache thrashing like leaf nodes. Hence, it is feasible for lock-free internal nodes to achieve high performance. We modify the internal nodes of ART as follows.

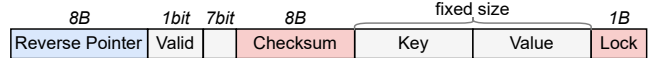
(1) **Homogeneous adaptive internal node.** As illustrated in Figure 2, a naive ART stores partial keys and child pointers separately. Such a heterogeneous design makes it hard to design a lock-free algorithm since the separated partial key and child pointer should be modified atomically. Besides, it incurs additional read amplification due to the inflexible fixed-sized internal nodes.

We come up with a homogeneous internal node design that embeds the partial keys into slots. First, this enables a child pointer to be modified together with its corresponding partial key atomically, laying the foundation for lock-free algorithms. Second, the read amplification can be reduced since internal nodes can have an arbitrary number of slots.

As shown in Figure 6a, an internal node of SMART consists of an 8-byte reverse pointer, several 8-byte slots, and an 8-byte header. The reverse pointer is used for cache validation, which will be presented in § 4.3. As for each slot, apart from the embedded 8-bit partial key and the 48-bit child pointer, we add a 1-bit *Leaf* field to indicate whether the pointer is pointing to a leaf node. When *Leaf* is set, a  $Len_{leaf}$  field is provided, which is used to support variable-sized keys (§ 4.5). When *Leaf* is unset, there is a 5-bit  $Type_{node}$  field to indicate



(a) The homogeneous adaptive internal node with the pessimistic 8-byte header.



(b) The update-in-place leaf node with the rear embedded optimistic lock.

Figure 6: The structure of the internal node and the leaf node in SMART. The reverse pointer and the in-header  $Type_{node}$  field are used for cache validation.

the type of the following internal node. Note that SMART mainly uses the  $Type_{node}$  to reduce the network bandwidth consumption rather than memory consumption. When fetching an internal node, SMART can RDMA\_READ only the required number of slots according to the  $Type_{node}$  field, reducing the read amplification and thus saving the network bandwidth.

(2) **Pessimistic 8-byte header of the internal node.** We choose the pessimistic method for path compression since both the optimistic and hybrid methods need two tree traversals to insert a nonexistent key. One entire tree traversal is required to search for the nonexistent key since not all compressed partial keys are stored in the header. The other traversal executes the actual insertion. In contrast, the pessimistic method can insert the nonexistent key through one traversal.

Besides, following previous designs [29, 33, 37], we fix the header size to 8 bytes, which can be changed atomically. If some partial keys overflow from the header, we store them in an empty following node. Although this may increase the tree height, we mitigate this with the help of cache (§ 4.3).

As shown in Figure 6a, a header consists of an 8-bit *Depth* field, a 5-bit  $Type_{node}$  field, a 3-bit  $Size_{array}$  field, and a 6-byte array of partial keys. The *Depth* field indicates the start position for matching the target key. The  $Type_{node}$  field is used for cache validation, which will be illustrated in § 4.3. The  $Size_{array}$  field records the length of the partial key array, where at most six partial keys can be stored.

**Lock-based leaf node.** In-place update schemes are preferred as it does not cause cache thrashing. To adopt the in-place update, lock-based concurrency control for the leaf node is required. This is acceptable since locks are fine-grained, as each leaf node in the radix tree only contains one key-value item. We design the leaf node structure as follows for concurrency control.

(1) **Checksum-based update-in-place leaf node.** The in-

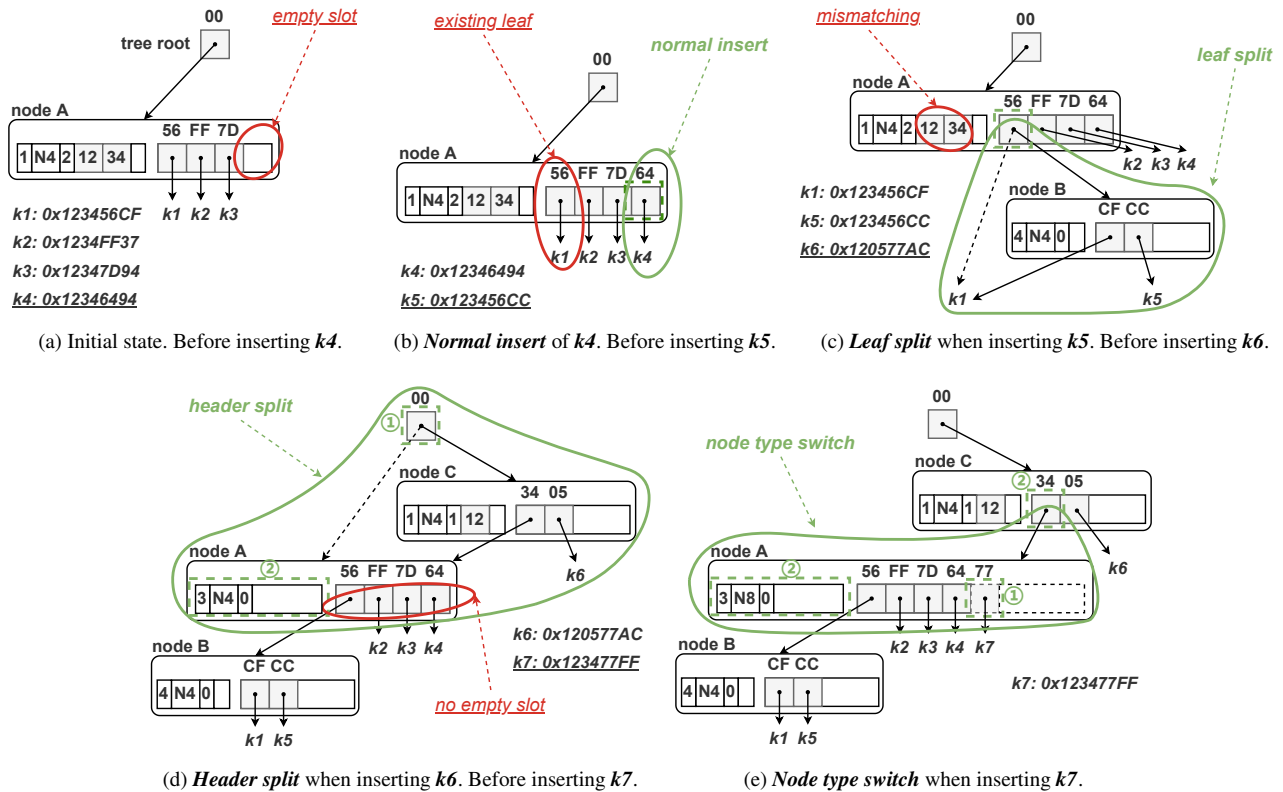


Figure 7: A step-by-step example of inserting several new keys into SMART with 8-bit partial keys. For clarity, hexadecimal partial keys are shown and reverse pointers are omitted. Each thick dotted box indicates an atomic CAS.

place update scheme overwrites the leaf node at the same address, causing conflicts among readers and writers. To avoid conflicts, we adopt an optimistic lock in each leaf node with a checksum-based consistency check mechanism [40, 53], where the fixed-sized key-value item in the leaf node is protected by a checksum. For write-write conflicts, an exclusive lock is used to synchronize the writers. As for read-write conflicts, when a writer modifies the leaf node, the checksum is re-calculated based on the new content of the leaf node and written with the new content. The readers verify the checksum after reading the leaf node. If the checksum verification fails, the reader conducts a re-read.

(2) *Rear embedded lock*. To further reduce the overhead of locks, we combine the lock release with the writing back of the updated leaf node by embedding the lock into each leaf node. Therefore, the two operations can be done via one single RDMA\_WRITE. Particularly, to avoid premature lock release, we ensure that the lock release is always triggered after the completion of writing back. We achieve this by placing the lock at the rear of a leaf node, which leverages the in-order delivery property of RNICs [12].

As shown in Figure 6b, a leaf node of SMART consists of an 8-byte reverse pointer, a *Valid* bit, an 8-byte checksum, a 1-byte rear lock and a fixed-sized key-value item. The reverse

pointer is used for cache validation, which will be illustrated in § 4.3. The *Valid* bit is used to indicate the deleted state.

#### 4.1.2 Concurrent Operations

Based on the above structural modifications, we demonstrate essential write-related sub-operations with a step-by-step example, as shown in Figure 7. Except for the in-place leaf update, all the sub-operations are lock-free. The complete operation process will be described in § 4.4.

*Normal insert*. During an insert, the target partial key may not be in the internal node yet. As shown in Figure 7b, after the WRITE of the new leaf node ( $k_4$ ), the client CASes the first empty slot in the node, together with the new partial key. If the CAS fails, the client checks whether the return value (*i.e.*, a new value of the slot written by a concurrent client) contains the target partial key. If yes, the client continues to traverse the tree following the return pointer. Otherwise, the client tries the insert again with the next empty slot.

*Leaf split*. If an existing leaf node is found during an insert, a leaf split is needed as shown in Figure 7c. Specifically, the client first calculates the rest of the longest common key prefix of the two leaf nodes ( $k_5$  and  $k_1$ ). Then it allocates sufficient sequentially-connected internal nodes to store the common key prefix in their headers. The last internal node will contain two child pointers pointing to the old and new leaf nodes. All

internal nodes and the new leaf node can be written in parallel, after which the client CASes the parent slot to point to the first new internal node. If the CAS fails, the client continues to traverse following the return pointer.

**Header split.** If a mismatching for in-header partial keys is found, a header split is required as shown in Figure 7d. Specifically, the client allocates a new NODE\_4 pointing to the split internal node and new leaf node ( $k_6$ ), with its header storing the matched part of partial keys. The new internal and leaf node can be written in parallel. Then the client CASes the parent slot to make it point to the new internal node (①). If CAS succeeds, the redundant in-header old partial keys are removed via an additional CAS (②). Otherwise, the client continues to traverse following the return pointer.

Note that the correctness of concurrent searches can be guaranteed by the in-header *Depth* value, which indicates the start position for matching the current key. A concurrent search READs the parent node and then the child node. Therefore, there are two situations of read-write conflicts. First, the READ of the parent node occurs after the CAS of the parent slot (①), while the READ of the child node occurs before the CAS of the split header (②). In this situation, redundant in-header partial keys are read, which does not affect the correctness. Second, the former READ occurs before the former CAS (①), while the latter READ occurs after the latter CAS (②). In this case, the reader re-reads the parent slot if finding partial keys missing according to the *Depth* value.

**Node type switch.** To avoid copy-on-write (COW) overhead and additional cache coherence introduced by out-of-place updates (**Challenge 1**), we conduct an in-place node type switch. This is feasible thanks to the homogeneous adaptive internal node design (§ 4.1.1). To be specific, we pre-allocate the contiguous space of NODE\_256 on MNs for each internal node. This consumes a little additional memory but enables lock-free operations during the node type switch. When neither a matching partial key nor an empty slot is found in the current internal node, the client can try to CAS the following empty slots one by one, whose addresses are behind the node (①) as shown in Figure 7e. After a successful CAS, the current best-fit node type can be determined by the index of the newly inserted slot. The client then tries to update the two old  $Type_{node}$  values (on the header and the parent slot) with the new one via two concurrent CASes (②), making the newly inserted leaf visible by subsequent search. If both CASes succeed or fail with return values containing  $Type_{node}$  values larger than/equal to the expected one, the node type switch is finished. Otherwise, the client retries the CASes.

**In-place leaf update.** To update a leaf node, the client first acquires the rear embedded lock in the leaf node. It then WRITES back the updated leaf node with the re-calculated checksum and the unset lock, after which the in-place leaf update is finished with the lock properly released.

### 4.1.3 RDMA-related Optimizations

To further optimize performance on DM, SMART adopts the following RDMA-related optimizations [23].

**Inline write.** For small-sized WRITE (e.g., writing internal nodes smaller than NODE\_16 or leaf nodes), the INLINE flag is set, enabling the RNIC to encapsulate payload into the work queue entry (WQE) and thus reducing PCIe overhead.

**Unsignaled verbs.** As for writing commands allowing asynchronous execution (e.g., CAS of the header during *header split*), SMART unsets the SIGNED flag to reduce the overhead of polling RDMA completion queues.

**Doorbell batching.** If a client issues multiple WQEs to the same queue pair (e.g., to the same MN), a doorbell batching is conducted to reduce PCIe overhead.

## 4.2 Read Delegation and Write Combining

SMART proposes the read-delegation and write-combining (RDWC) technique on DM to eliminate inter-client redundant I/Os in terms of reads and writes, respectively, to break through the IOPS upper bound.

**Hash-based local locks.** The inter-client redundant I/Os on each CN occur among the concurrent read and write operations on the same key or address. Therefore, computing-side local locks are needed to collect the concurrent operations.

We maintain the local locks in each CN as a table, similar to the local lock table of HOCL in Sherman [53]. However, unlike Sherman, which maintains each local lock for a coarse-grained global lock, SMART maintains each local lock for a key (i.e., fine-grained leaf node). It is challenging to store all such locks in each limited computing-side memory. To address this, we use hash-based local locks, where a lock corresponds to a set of keys with the same hash value.

We dynamically maintain a *unique key* in each local lock to solve the hash-conflict problem of our hash-based scheme. Specifically, the first client who acquires a local lock successfully will record its target key as the unique key of this local lock. The subsequent clients who fail to acquire this local lock will conduct a hash-conflict check by comparing their target key with the unique key. If the target key is exactly the same as the unique key, the client can be involved in the read delegation or write combining. Otherwise, a hash conflict is found, and the client should execute a normal remote read or write on its own for correctness. The unique key is freed when the first client releases the local lock.

**Read delegation.** To reduce inter-client redundant I/Os for reads, a *delegation client* can be elected on each CN to execute the same read, and then share its RDMA\_READ result with other waiting clients. The first client who acquires the local lock successfully is the delegation client and the subsequent clients who fail to acquire the lock are the waiting clients. The relationship between the delegation client and the waiting clients is similar to that between the first cache miss and the subsequent delayed cache hits in the cache system [5].

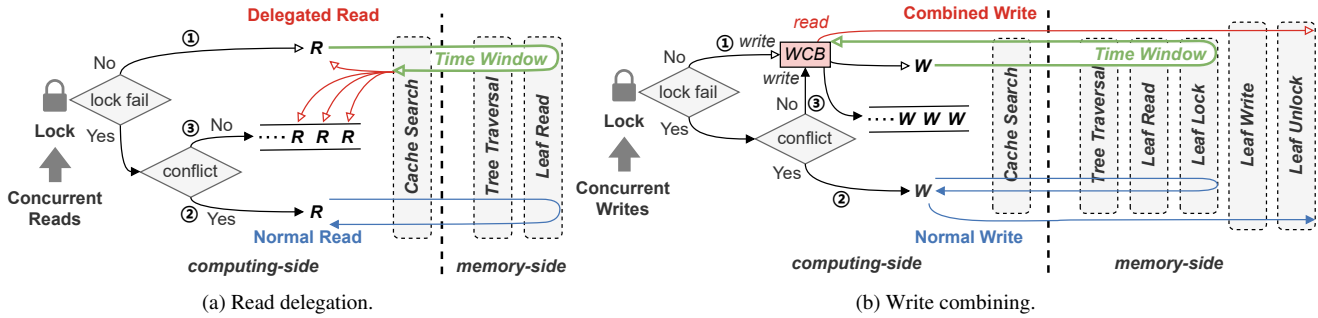


Figure 8: The processes of the read delegation and the write combining on SMART respectively.

We implement this as shown in Figure 8a. After acquiring the corresponding local lock successfully, the delegation client records its target key as the unique key and then conducts the remote tree search (*i.e.*, including cache search, tree traversal, and leaf node read), which is the time window of read delegation (①). During the time window, the subsequent clients failing to acquire the local lock first execute the hash-conflict check by comparing their target key with the unique key. If a hash conflict is found, the client executes a normal tree search by itself (②). Otherwise, it pushes itself into a read-waiting queue and waits for the search result from the first client (③). Finally, the delegation client shares its search result with the waiting clients and releases the local lock.

**Write combining.** Write combining (WC) is a normal technology in modern processors [11]. When a processor intends to issue multiple writes to the same memory region in a small time window, it combines the writes into a single burst write so as to save the system bus bandwidth. This idea, also known as write coalescing, is applied to many storage systems [22, 28, 50]. Inspired by this, we find it feasible to conduct a WC on each CN. When clients intend to make several concurrent key-value writes to the same memory-side key or address, they can combine the writes into a single consensus write so as to save the network bandwidth and the limited IOPS of RNICs.

We implement WC on DM as shown in Figure 8b. A client that succeeds in acquiring the corresponding local lock first records its target key as the unique key and writes its new value into the write combining buffer (WCB), and then conducts the remote tree insert or update (①). Differently, the time window of write combining is the former partial period of tree insert or update (*i.e.*, cache search, tree traversal, and lock acquirement on leaf node). After that, the client reads the combined consensus result from WCB and then makes a RDMA\_WRITE to write back the result and release the remote lock. Finally, the client releases the local lock. During the write-combining time window, the subsequent clients first perform the same hash-conflict check. If a hash conflict is found, the client performs a normal tree insert or update on its own (②). Otherwise, it first writes its expected value into the

WCB (with local lock-based concurrency control), making the value visible to the first client. Then the client pushes itself into a write-waiting queue to wait for the completion of the remote write (③).

**Put both together.** Naively putting read-delegation and write-combining together may introduce incorrect read results when a client reads a key-value item after writing it. Specifically, the latter read may be delegated by a client whose read happens before the write operation. In this case, the old value (*i.e.*, the value of the item before the client’s write) is returned to the read operation that happens after the write, breaking the causality of the read and write. We use the same time window for read-delegation and write-combining to address this issue. In this way, the write and read operations with causal relations are included in two non-overlapped time windows, and thus, the above issue can be avoided. To achieve this, we let readers and writers operating on the same key fairly acquire the same local lock, where the winner decides the time window. Each local lock is associated with two waiting queues, *i.e.*, a read queue and a write queue, so as to conduct read delegation and write combining exclusively and concurrently. In our implementation, 4M 32-bit local locks are sufficient on each CN, consuming only nearly 3% of cache size.<sup>2</sup>

### 4.3 ART Cache

**ART-indexed cache.** To reduce remote access during tree traversal, a memory-efficient ART-indexed cache is designed on each CN to store partial internal nodes of SMART. To be specific, utilizing the feature that each radix tree node (excluding header) can be uniquely identified by a key prefix, we adopt a local ART on each CN to index the cached internal nodes. As shown in Figure 9, each leaf node (*i.e.*, cache entry) of the local ART contains the snapshot of a traversal context (*i.e.*, the content of an internal node being read from MNs, the *Depth* value, and the address of the node).

**Cache invalidation situations.** Since we cache the slots of the internal nodes in clients, changing the slots in the disaggregated memory leads to cache invalidation. We analyze all

<sup>2</sup>Note that with  $N$  clients in each CN, there are at most  $N$  dynamically-allocated WCBs and unique keys at the same time, whose memory consumption (*i.e.*, size of  $N$  key-value items) is negligible.



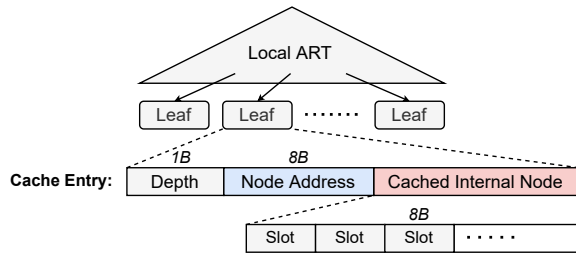


Figure 9: The structure of the ART cache.

operations that change the slots (*i.e.*, slot insert, update, and delete) and find there are only three types of cache invalidation in the current SMART design, *i.e.*, Type 1: *adjustments on the parent-child relationship*, Type 2: *node type changes*, and Type 3: *deleted nodes*. Specifically:

**For slot insert**, inserting a new slot does not affect the client cache since the new slot is not in the client cache.

**For slot update**, it contains four situations according to the structure of slots in Figure 6a (note that the *Partial Key* field keeps unchanged until deleted):

- Updating the *Child Pointer* field. This type of cache invalidation corresponds to Type 1.
- Updating the  $Type_{node}$  field. This type of cache invalidation corresponds to Type 2.
- Updating the *Leaf* field. Since leaf nodes have different addresses from internal nodes, the *Leaf* field update should be combined with a *Child Pointer* update. Thus this type of cache invalidation corresponds to Type 1.
- Updating the  $Len_{leaf}$  field. This field keeps unchanged since SMART is currently designed for fixed-sized leaf nodes. The support for variable-sized leaf nodes will be discussed in § 4.5.

**For slot delete**, this type of cache invalidation corresponds to Type 3.

**Reverse check mechanism.** To handle the above three types of cache invalidation situations, we design a reverse check mechanism specifically for SMART, as existing solutions on B+ trees are infeasible for ART. We store the check information in remote internal and leaf nodes. A mismatch between check information and cache content indicates an outdated cache entry, which will be invalidated.

**(1) Adjustments on the parent-child relationship.** We store a reverse pointer in the front of each node to point to its parent, as shown in Figure 6. If the client reads a remote node according to a cached pointer, it checks whether the reverse address is equal to the node address in the cache entry. If not, a mismatch is found, which indicates that a newly inserted node (*e.g.*, caused by *leaf split* or *header split*) is invisible to the client due to the outdated cache entry.

**(2) Node type changes.** We design a  $Type_{node}$  field in the header of each node to indicate the current type of the node, as shown in Figure 6a. If the client reads a remote node according to a cached pointer, it checks whether the in-header

$Type_{node}$  value being read is the same as that in the cached slot. If not, and the in-header  $Type_{node}$  value is larger than the cached one, read the rest of the remote node.

**(3) Deleted nodes.** We set the in-header  $Type_{node}$  value to zero to indicate the deleted state of an internal node. As for a deleted leaf node, the *Valid* bit is unset.

## 4.4 Operations

All operations first search in the cache for the deepest slot that is matched by the prefix of the target key. If none of the cached slots hits, start the traversal from the tree root slot.

**Search.** The client first reads the node according to the slot, after which a *reverse check* is conducted to check if the cache entry expires. If yes, invalidate the cache entry and retry this search. As for a leaf node being read, the target item is found if its key is the same as the target key. Otherwise, it does not exist. As for an internal node, if all the in-header partial keys are matched, and the next target partial key can be found in a slot, read the next node along the child pointer in the slot and repeat the process. Otherwise, the target item does not exist.

**Insert/Update.** The client first reads the node and conducts a *reverse check* like the search. After that, as for a leaf node, if its key is the same as the target key, execute an *in-place leaf update*. Otherwise, a *leaf split* is needed. As for an internal node, if a mismatching for the in-header partial keys is found, conduct a *header split*. Otherwise, turn to search among the slots. If the current target partial key can be found in a slot, read the next node along the corresponding child pointer in the slot and start the process again. Otherwise, conduct a *normal insert* with the next empty pointer slot. If no empty slot can be found, a *node type switch* is needed.

**Delete.** Delete operations have a similar process as insert operations. A normal delete clears the slot pointing to the target leaf node via RDMA\_CAS and unsets the *Valid* bit of the deleted leaf node. Opposite operations of *leaf split* and *header split* are conducted for path compression.

**Scan.** At each level of traversal, the client conducts parallel RDMA\_READs to fetch all nodes inside the target key range. For each RDMA\_READ, the client processes the node being read in the same way as the search operation, with an additional comparison between partial keys and target key range to exclude unwanted concurrent search paths. Like many other existing tree indexes [53, 59] on DM, SMART does not guarantee the scan is atomic with concurrent insert or update operations.

## 4.5 Discussion

**Support for variable-sized keys and values.** SMART currently supports fixed-sized keys and values. For variable-sized keys and values, the optimizations of *update-in-place leaf node* and *rear embedded lock* in SMART are no longer applicable. Instead, SMART can use the RCU scheme to out-of-place update the leaf node to support variable-sized keys and values. The search, insert and delete operations on

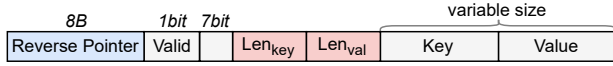


Figure 10: The structure of the variable-sized leaf node.

variable-sized key-value items are the same as that on fixed-sized ones.

As for the leaf node structure, SMART can follow the design in RACE [60]. As shown in Figure 10, the leaf node structure includes a  $Len_{key}$  field and a  $Len_{val}$  field, which indicate the sizes of the following *Key* and *Value* fields, respectively. SMART can use the 7-bit  $Len_{leaf}$  field in the parent slot and a pre-configured  $length\_unit$  value to indicate the length of the leaf node. The maximum length of a leaf node is  $2^7 \cdot length\_unit$ . When a key-value item exceeds the maximum length, SMART can store the remaining content in a second key-value block linked to the leaf node.

Moreover, the cache validation mechanism (§ 4.3) can be extended to support variable-sized leaf nodes with a new cache invalidation situation, *i.e.*, Type 4: *leaf node length changes*. When a client reads a remote leaf node according to a cached slot, it checks whether the sum of the  $Len_{key}$  and  $Len_{val}$  values equals the  $Len_{leaf} \cdot length\_unit$  value. If not, the cached slot is invalid.

**Generality of techniques in SMART.** Some techniques in SMART can also be applied to other kinds of indexes. Particularly: 1) The *RDWC* technique can benefit any tree indexes since it is transparent to the lower-level index structures. When applied to other index structures, it brings about the same performance improvement as applied to ART. 2) The *reverse check mechanism* can benefit any radix-tree-based indexes. It is designed to handle the cache validation problems caused by ART’s features. 3) The *rear embedded lock* can be adopted in any lock-based structures on DM to save one RTT.

**The first lock-free ART design.** A pure lock-free ART can be formed with the lock-free node design in Figure 6a and a lock-free leaf node design with a traditional RCU scheme. To our knowledge, this is the first lock-free ART design. In our implementation, SMART can degenerate into the pure lock-free ART by disabling the optimizations of *update-in-place leaf node* and *rear embedded lock*.

## 5 Evaluation

### 5.1 Experimental Setup

**Testbed.** We run all experiments on 16 physical machines (16 CNs and 2 MNs)<sup>3</sup> on the Clemson cluster of Cloud-Lab [13]. Each machine has two 36-core Intel Xeon CPUs, 256GB of DRAM, and one 100Gbps Mellanox ConnectX-6 IB RNIC. Each RNIC is connected to a 100Gbps Ethernet switch. Each MN owns 64GB DRAM and 1 CPU core for network connection and memory allocation. Each CN owns

<sup>3</sup>Like Sherman [53], we make two physical machines act as both CN and MN to save machine resources.

4GB DRAM and 64 CPU cores, where each core can serve as a client. The MNs register memory with huge pages to reduce page translation cache misses of RNICs [12].

**Workloads.** Without explicit mention, we use the index microbench [55] to generate YCSB [10] workloads like previous work [6, 26, 39]. We evaluate SMART with 6 YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (100% read), D (latest-read, 95% read, 5% insert), E (95% scan accessing up to 100 items, 5% insert) and an additional LOAD (100% insert) workloads, using the default Zipfian distribution for all workloads except for YCSB LOAD and D. For most workloads, we test 2 key types, *i.e.*, integer (8-byte) and string (32-byte). For string workloads, we use 125 million publicly available email addresses [15] and conduct a common pre-processing (*i.e.*, swap username and domain fields of email addresses) like previous work [32, 38, 39, 55]. We use 8-byte values consistent with prior work [6, 24, 38, 41, 53, 56]. For each workload, we populate 60 million keys before conducting 60 million operations, except for the LOAD test.

**Comparisons.** We compare SMART with two state-of-the-art tree indexes, *i.e.*, Sherman [53] and ART [32]. We use the default configuration of Sherman (*e.g.*, a span size of 32 for long key) with all optimizations enabled (*e.g.*, on-chip memory). Since ART is not designed for DM, we port it to DM by re-implementing it from scratch (as mentioned in § 3), including its synchronization design (*i.e.*, ROWEX [33]). For better baseline performance, we apply the HOCL of Sherman to ART and any other baselines of SMART. Coroutines are used in each client to hide RDMA polling overhead.

### 5.2 Performance Comparison

Figures 11 and 12 present the throughput-latency curves of the three indexes with integer and string keys respectively, using various numbers of clients (16 at least and 896 at most, evenly distributed across 16 CNs). Without loss of generality, we discuss the performance of integer keys in the following.

**Search-only workload (YCSB C).** For the YCSB C workload, SMART outperforms Sherman by  $2.8\times$  due to no leaf read amplification, as mentioned in § 3. Moreover, it outperforms ART by  $1.2\times$  due to the read delegation mechanism for reducing redundant I/Os. It is worth noting that SMART achieves up to 96M requests per second, which breaks through the total IOPS upper bound of memory-side RNICs (about 90 Mops in total with the two MNs). This is because the read delegation can perform concurrent duplicated reads with only one delegated read. Besides, the similar P99 latency of SMART and ART shows that the read delegation causes near-zero overhead.

**Insert workload (YCSB LOAD, D).** For the YCSB LOAD workload, SMART outperforms Sherman and ART by  $1.6\times$ ,  $1.5\times$  in throughput and achieves  $1.4\times$ ,  $1.5\times$  lower P99 latency respectively. This can be attributed to the design of the lock-free internal nodes. Specifically, both Sherman and ART have low throughput and high latency due to the node-

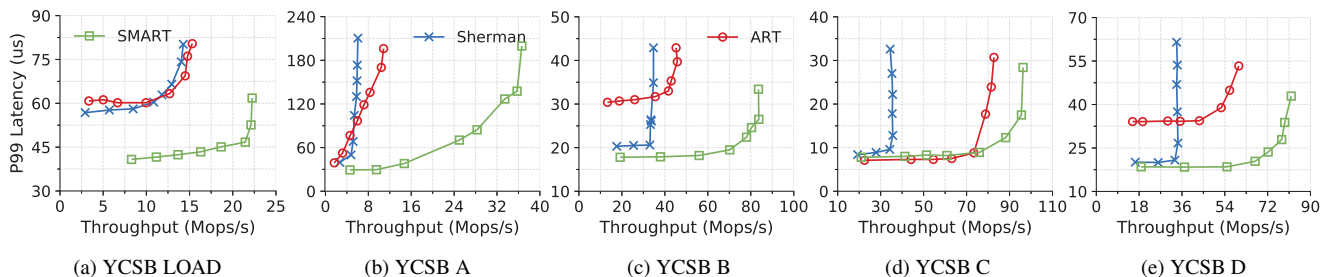


Figure 11: The performance comparison of tree indexes on DM under YCSB workloads of integer keys.

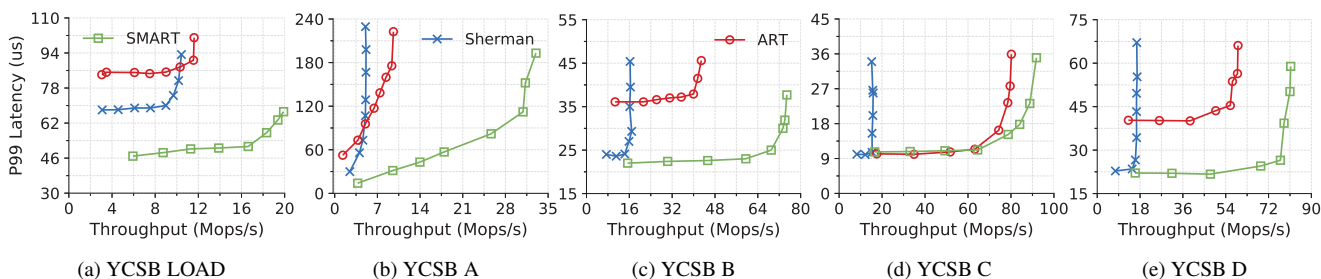


Figure 12: The performance comparison of tree indexes on DM under YCSB workloads of string keys.

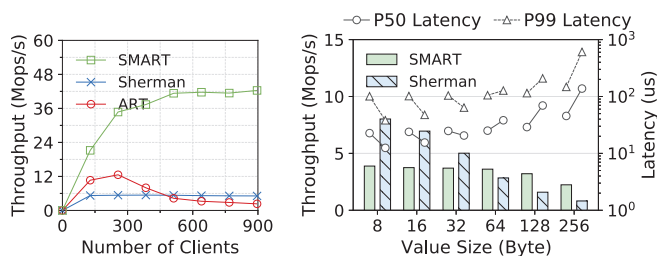


Figure 13: The scalability of Figure 14: The performance of scan untree indexes under the YCSB E workload of integer keys with different value sizes.

grained locks, which introduce additional RTTs with frequent lock-fail retries, thus wasting the limited IOPS of RNICs in write-intensive scenarios (*i.e.*, 50% insert). Interestingly, with string workloads, the latency of ART becomes much worse since the smaller set of string partial keys (*e.g.*, alphanumeric characters) aggravates concurrency conflicts.

For the YCSB D workload, SMART achieves  $2.4\times$  and  $1.4\times$  higher throughput and  $1.1\times$  and  $1.8\times$  lower P99 latency, compared with Sherman and ART respectively. With fewer write conflicts (*i.e.*, only 5% insert), read and write amplifications become the main reason for the poor performance of Sherman. ART still has a high tail latency since concurrent writes cause cache misses, leading to remote tree traversals and thus continuous lock operations on the remote tree.

**Update workload (YCSB A, B).** Compared with Sherman and ART, SMART gains  $6.1\times$  and  $3.4\times$  improvement

in throughput and  $1.4\times$  and  $1.3\times$  reduction in latency for YCSB A, and achieves  $2.4\times$  and  $1.8\times$  higher throughput and  $1.1\times$  and  $1.7\times$  lower P99 latency for YCSB B, respectively.

Unlike the insert workload, YCSB A and B follow a Zipfian distribution of skewness 0.99, indicating a high amount of update concurrency conflicts. Consequently, Sherman performs poorly with YCSB A due to its coarse-grained, lock-based concurrency control. ART performs better than Sherman since update operations do not modify the partial key fields and thus do not need to acquire locks. However, the out-of-place update scheme used by ART causes cache thrashing, resulting in huge cache-miss overhead and thus much higher latency than SMART. Note that the cache thrashing also impacts search performance, leaving a poor performance of ART on YCSB B (with only 5% update). As shown in Figure 13, ART experiences performance collapse with increasing clients due to severe cache thrashing. In contrast, SMART shows excellent scalability due to the cache-friendly in-place leaf node design and fine-grained concurrency control.

**Scan workload (YCSB E).** We evaluate the performance of scan operations with 128 clients using varying value sizes as shown in Figure 14. For a small value size (*e.g.*, 8 bytes), SMART shows poorer performance than Sherman since the small-sized leaf nodes saturate the memory-side IOPS upper bound, which is an inherent shortcoming of radix trees. However, for a value size larger than 64 bytes, which is common in real-world workload [4, 58], the scan performance of Sherman becomes worse than SMART since the large-sized leaf nodes rapidly saturate the bandwidth bottleneck.



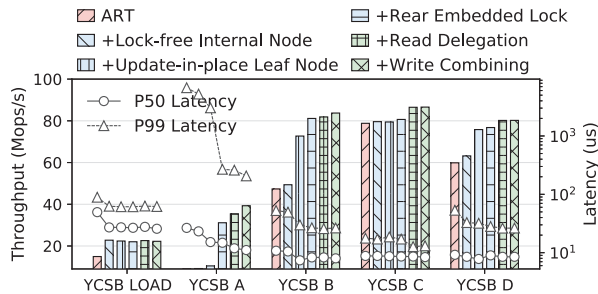


Figure 15: The factor analysis of overall performance on SMART.

### 5.3 Factor Analysis for SMART Design

Figure 15 presents the factor analysis on SMART. We start with the naive ART and apply each proposed technique one by one. We use 16 CNs (each launches 24 clients) and integer keys for experiments in this section.

**+ Lock-free internal node.** The lock-free internal nodes mainly contribute to the insert workload. With YCSB LOAD, it brings  $1.5\times$  improvement in throughput and  $1.8\times/1.4\times$  reduction in P50/P99 latency. Unlike ROWEX, lock-free internal nodes eliminate expensive lock overhead during insertion and thus improve performance.

**+ Update-in-place leaf node.** In-place update scheme mainly contributes to the update workload. It achieves  $1.5\times$  improvement in throughput and  $1.4\times/1.7\times$  reduction in P50/P99 latency with YCSB B. The in-place update scheme alleviates the cache coherence problem, as the addresses of the cached leaf nodes never expire until being deleted.

**+ Rear embedded lock.** The rear embedded locks further optimize the in-place update scheme. It eliminates the lock-releasing overhead, saving one RTT during each update. With YCSB A, it improves throughput by  $3.0\times$  and reduces tail latency by  $11.3\times$ .

**+ Read delegation.** The read delegation mechanism contributes to the search workload. It brings  $1.1\times$  throughput improvement and  $1.3\times$  tail latency reduction with YCSB C. It eliminates superfluous reads and thus saves network I/O consumption, so as to support more client requests.

**+ Write combining.** The write combining mechanism mainly contributes to the write-intensive workload. It improves the throughput by  $1.1\times$  and reduces tail latency by  $1.3\times$  with YCSB A.

As the RDWC technique can reduce concurrency conflicts similar to HOCL, we compare their efficiency by applying them on SMART respectively. As shown in Figure 16, when applying the primitive HOCL design, SMART shows poor performance with an average of 0.76 lock-fail retry count, due to the limited on-chip memory space (128MB per RNIC in our evaluation) with only 2 MNs, which is insufficient for a large number of fine-grained locks. With E-HOCL (*i.e.*, integrating the rear embedded lock technique into HOCL), SMART achieves much better performance with an average

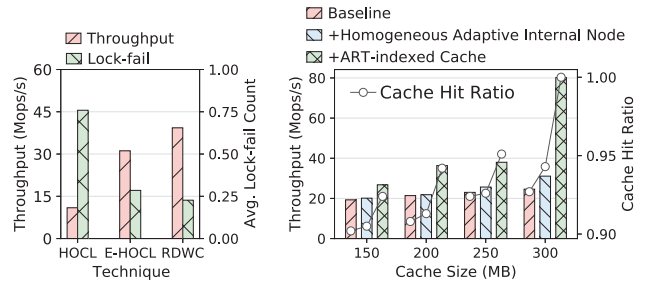


Figure 16: The efficiency comparison of HOCL, E-HOCL and RDWC under the YCSB A workload. Figure 17: The factor analysis of cache efficiency on SMART under the YCSB A workload.

of 0.29 lock-fail retry count. However, despite the optimization, HOCL still shows lower improvement efficiency than RDWC, which can introduce a 26.2% higher throughput. This is because RDWC saves not only the lock overhead but also the superfluous bandwidth consumption of reads and writes.

As the design of RDWC is transparent to the lower-level index structures, it will lead to the same amount of performance improvements on Sherman, *i.e.*,  $1.3\times$  and  $1.1\times$  under write-intensive and read-only workloads (Figure 15). Therefore, after applying RDWC to Sherman, SMART can still achieve  $4.7\times$  ( $= 6.1/1.3$ ) higher throughput under write-intensive workloads and  $2.5\times$  ( $= 2.8/1.1$ ) higher throughput under read-only workloads.

**Cache-related techniques.** Some cache-related techniques contribute to cache efficiency: **1) Homogeneous adaptive internal node.** Due to the homogeneous adaptive internal node design, more fine-grained and flexible adaptive nodes are available, saving cache space with smaller sizes of cached nodes. **2) ART-indexed cache.** Compared with a normal hash-based cache index, ART-indexed cache can efficiently save memory consumption of index keys without redundant key prefixes stored. As shown in Figure 17, after applying the above two techniques one by one, SMART achieves an increasing cache hit ratio and overall throughput under each specific limited cache size.

### 5.4 Sensitivity

In this section, we investigate how the workload skewness, key size, and value size affect the performance of SMART. We use 16 CNs with 16 clients each and integer keys for the sensitivity evaluation.

**Skew test.** Figure 18a shows the performances of different tree indexes on a generated Zipfian workload [35] (50% search + 50% update) with various skewness. SMART performs best under both slightly and highly skewed workloads. Sherman shows a good performance in slightly skewed workloads, while having the poorest performance in highly skewed workloads because of its coarse-grained lock-based concurrency control design. ART performs better than Sherman in



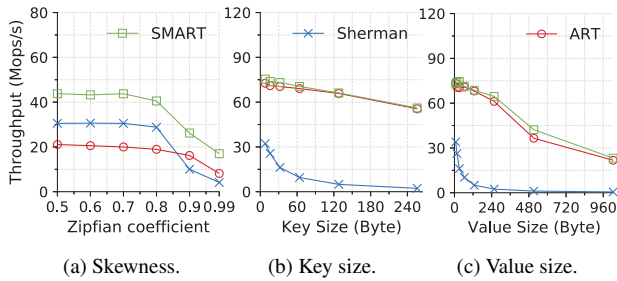


Figure 18: The sensitivity analysis.

highly skewed workloads due to the lock-free RCU scheme but performs worst in slightly skewed workloads due to cache thrashing. Note that the RDWC in SMART does not benefit the overall throughput since the network bandwidth is unsaturated. As the Zipfian skewness grows from 0.5 to 0.99, the performance of ART and SMART decrease by the same multiple ( $2.6\times$ ), and thus their performance gap is reduced. The performance of Sherman decreases by  $7.4\times$ , indicating the poor efficiency of coarse-grained lock-based design.

**Impact of key/value size.** Figures 18b and 18c show the impact of key size and value size on the performances of the three tree indexes under YCSB C with sufficient caches. As the key size grows from 8 to 256 bytes, SMART and ART show a slight performance decline ( $1.3\times$ ), while Sherman experiences a rapid drop in performance ( $14\times$ ). As the value size grows from 8 to 1024 bytes, the performance declines of SMART, ART and Sherman are  $3.1\times$ ,  $3.4\times$  and  $64\times$ , respectively. This is because, during each search, Sherman needs to fetch the whole leaf node, whose size grows with key and value size, causing the rapidly increasing consumption of network bandwidth. On the contrary, SMART and ART only need to fetch the fine-grained small-sized leaf node. Thus, they are not bounded by the network bandwidth bottleneck, showing a stable performance with varying key sizes and value sizes. The performances of ART and SMART are close since the read delegation in SMART does not benefit the throughput under the unsaturated network. This is consistent with the results shown in Figure 11d.

## 6 Related Work

**Disaggregated Memory.** The DM architecture is widely discussed in the literature [3, 9, 16, 19, 20, 27, 47], which is proposed to address the problem of a growing imbalance between computing and memory resources. Many recent academic works have been conducted on DM. LegoOS [46] designs a distributed operating system for disaggregated resource management. PolarDB Serverless [8] co-designs the database and DM to achieve better dynamic resource provisioning and faster failure recovery speed. Clover [52] explores an efficient manner to build a key-value store on disaggregated persistent memory (PM), with careful designs between the data plane and the metadata/control plane. FUSEE [48]

designs a fully memory-disaggregated key-value store that brings disaggregation to metadata management. ROLEX [34] proposes a scalable RDMA-oriented learned key-value store that dissociates the model retraining from data modification operations. RACE [60] is an extendible RDMA-based hashing index with lock-free remote concurrency control and efficient remote resizing. Sherman [53] is a B+ tree index on DM with RDMA-friendly software techniques to boost index write performance. SMART focuses on building a fast, scalable radix tree index on DM with small read and write amplifications.

**RDMA-based Tree Indexes.** Attracted by the high performance of RDMA, there are increasing studies focusing on RDMA-based tree indexes [1, 41, 45, 53, 59]. Many studies conduct operations via remote procedure calls (RPCs), which is unsuitable for DM due to weak memory-side computation power. FG [59], designed as a B-link tree, is the first index that completely leverages *one-side verbs* for write operations and thus supports DM. Sherman [53] is the state-of-the-art B+ tree index with several RDMA-friendly software techniques. However, constrained by the structure of the B+ tree, it suffers from low peak throughput and early latency deterioration due to read and write amplifications. Besides, extending RDMA interfaces is another approach to design tree indexes on DM, which offloads index write operations into memory-side NICs via SmartNICs or other customized hardware [1, 7, 14, 25, 36, 44, 49]. To our knowledge, SMART is the first radix tree index on DM that achieves high performance with commodity RNICs.

## 7 Conclusion

Based on a thorough theoretical and experimental analysis of tree indexes built on DM, this paper points out the performance bottleneck of B+ trees on DM due to severe read and write amplifications and then presents SMART, the first radix-tree-based index on DM. SMART addresses the challenges of applying ART on DM, including a hybrid concurrency control scheme to reduce lock overhead and avoid cache thrashing, a read-delegation and write-combining technique to reduce redundant I/Os, and a tailed cache validation mechanism. Our evaluation results show that SMART outperforms the state-of-the-art B+ tree on DM by up to  $6.1\times$  under write-intensive workloads and  $2.8\times$  under read-only workloads.

## Acknowledgments

We sincerely thank our shepherd Steven Swanson and the anonymous reviewers for their constructive comments and suggestions. This work is supported by the National Natural Science Foundation of China (Project No. 61971145), the Natural Science Foundation of Shanghai (Project No. 22ZR1407900), and Huawei Cloud. Yangfan Zhou (zyf@fudan.edu.cn) and Pengfei Zuo (pfzuo.cs@gmail.com) are the corresponding authors.

## References

- [1] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 120–126. ACM, 2019.
- [2] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proc. VLDB Endow.*, 1(1):598–609, 2008.
- [3] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020.
- [4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.
- [5] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 495–513. ACM, 2020.
- [6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM, 2018.
- [7] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA interface for distributed systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 228–242. ACM, 2021.
- [8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. PolarDB Serverless: A cloud native database for disaggregated data centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.
- [9] Amanda Carbonari and Ivan Beschastnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 164–170. ACM, 2017.
- [10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.
- [11] Intel Corporation. Write combining memory implementation guidelines. <https://download.intel.com/design/PentiumII/applnots/24442201.pdf>.
- [12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.
- [13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.
- [14] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohita, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66. USENIX Association, 2018.
- [15] Fonxat. 300 million email database. <https://archive.org/details/300MillionEmailDatabase>, 2018.

- [16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.
- [17] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.
- [18] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiyang Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 417–433. ACM, 2022.
- [19] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, pages 10:1–10:7. ACM, 2013.
- [20] Eric Hooper. Intel rack scale design: Just what is it? <https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it>, 2018.
- [21] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An adaptive B<sup>+</sup>-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.
- [22] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: Remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 253–268. USENIX, 2003.
- [23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.
- [24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.
- [25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 756–771. ACM, 2021.
- [26] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A high performance persistent range index using PAC guidelines. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 424–439. ACM, 2021.
- [27] HP Labs. The machine: A new kind of computer. <https://www.hpl.hp.com/research/systems-research/themachine>, 2014.
- [28] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 2–13. ACM, 2009.
- [29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 257–270. USENIX Association, 2017.
- [30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 462–477. ACM, 2019.
- [31] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.
- [32] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

- [33] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.
- [34] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 99–114. USENIX Association, 2023.
- [35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014.
- [36] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 318–333. ACM, 2019.
- [37] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 1–16. USENIX Association, 2021.
- [38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM, 2012.
- [39] Ajit Mathew and Changwoo Min. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, 2020.
- [40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.
- [41] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed B-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 451–464. USENIX Association, 2016.
- [42] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a data-center. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.
- [43] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.
- [44] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.
- [45] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.
- [46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.
- [47] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Disaggregating and consolidating network functionalities with SuperNIC. *CoRR*, abs/2109.07744, 2021.
- [48] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.
- [49] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarini, and Gustavo Alonso. StRoM: Smart remote memory. In *EuroSys '20: Fifteenth EuroSys Conference 2020*,



*Heraklion, Greece, April 27-30, 2020*, pages 29:1–29:16. ACM, 2020.

- [50] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 101–114. USENIX, 2010.
- [51] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 1–15. ACM, 2017.
- [52] Shin-Yeh Tsai, Yizhou Shan, and Yiyang Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.
- [53] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed B+tree index on disaggregated memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1033–1048. ACM, 2022.
- [54] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.
- [55] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huan Chen, Michael Kaminsky, and David G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 473–488. ACM, 2018.
- [56] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.
- [57] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree based indexing for cloud data processing. *Proc. VLDB Endow.*, 3(1):1207–1218, 2010.
- [58] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.
- [59] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 741–758. ACM, 2019.
- [60] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

## A Artifact Appendix

### Abstract

The artifact provides the source code of SMART and automated scripts to reproduce all the experiment results in the paper. The experiment results can show the superiority of ART on DM compared with the B+ tree and demonstrate the efficacy and efficiency of SMART we design.

### Scope

**Superiority of ART on DM.** By reproducing the experiments of Figure 3, the artifact can validate that the radix tree is more suitable for DM than the B+ tree due to smaller read amplification under *read-only workloads*.

**Challenges of ART on DM.** By reproducing the experiments of Figure 4, the artifact can validate that ART suffers from significant challenges on DM under *hybrid read-write workloads*.

**Efficacy and Efficiency of SMART.** By reproducing the experiments of Figure 11-18, the artifact can validate that SMART can show better performance under YCSB workloads, compared with the state-of-the-art B+ tree on DM and a naive ART design.

### Contents

**Source codes.** The artifact contains source codes of SMART and the compared baselines (*e.g.*, ART). Specifically, the source code of SMART contains the implementation of our three key designs, *i.e.*, the hybrid ART concurrency control scheme, the read-delegation and write-combining technique, and the reverse check mechanism for cache validation.

**Automated scripts.** The artifact also contains automated scripts to reproduce all the experiment results in the paper, *i.e.*, Figure 3-4, 11-18. Each figure has a Python script to automatically reproduce and visualize the experimental results.

### Hosting

The artifact is available at <https://github.com/dmemsys/SMART>. Please use the *latest* commit version on the *main* branch.

### Requirements

The artifact is developed and tested using the r650 machines on CloudLab. 16 r650 machines are needed to reproduce all the results. Each r650 machine has two 36-core Intel Xeon CPUs, 256GB of DRAM, and one 100Gbps Mellanox ConnectX-6 IB RNIC. Each RNIC is connected to a 100Gbps Ethernet switch.

### Tutorial

**Environment setup.** To set up the environment, please clone the source codes to the r650 machines. The necessary dependencies can be installed using our provided shell scripts in the artifact. Listing 1 shows the commands to set up the experiment environment.

Listing 1: Commands to set up the environment.

```
1 # Get the source codes
2 git clone https://github.com/dmemsys/SMART
3 git clone https://github.com/River861/Sherman
4 # Set bash as the default shell
5 sudo su && chsh -s /bin/bash
6 # Install Mellanox OFED
7 cd SMART
8 sh ./script/installMLNX.sh
9 # Resize disk partition
10 sh ./script/resizePartition.sh
11 reboot
12 sudo su && resize2fs /dev/sda1
13 # Install libraries and tools
14 cd SMART
15 sh ./script/installLibs.sh
16 # Setup hugepages
17 echo 36864 > /proc/sys/vm/nr_hugepages
```

**Workloads generation.** The index microbench is used to generate YCSB workloads, including two key types, *i.e.*, integer and string. Listing 2 shows the commands to generate all the workloads to reproduce the results.

Listing 2: Commands to generate all workloads.

```
1 # Download YCSB source code
2 cd SMART/ycsb
3 sudo su && curl -O --location https://github.
4   com/brianfrankcooper/YCSB/releases/
5   download/0.11.0/ycsb-0.11.0.tar.gz
6 tar xfvz ycsb-0.11.0.tar.gz
7 mv ycsb-0.11.0 YCSB
8 # Download the email dataset
9 gdown --id 1ZJcQOuFI7IpAG6ZBgXwhjEeK01T7Alzp
# Start to generate all the workloads
sh generate_full_workloads.sh
```

**Results Reproduced.** The artifact provides a single batch script to reproduce all the experiments. This script should be run on a *master node*, which can directly establish SSH connections to other nodes of the r650 cluster.

To reproduce the experiments, please set up the *home\_dir* and *master\_ip* values in *./exp/params/common.json*. Then the script can be run. Listing 3 shows the commands. The reproduced results will be saved automatically.

Listing 3: Commands to start all experiments.

```
1 sudo su && cd SMART/exp
2 # Run all the experiments
3 sh run_all.sh
```





# ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities

Vasily A. Sartakov  
*Imperial College London*

Lluís Vilanova  
*Imperial College London*

Munir Geden  
*Imperial College London*

David Eyers  
*University of Otago*

Takahiro Shinagawa  
*The University of Tokyo*

Peter Pietzuch  
*Imperial College London*

## Abstract

Cloud environments host many tenants, and typically there is substantial overlap between the application binaries and libraries executed by tenants. Thus, memory de-duplication can increase memory density by allocating memory for shared binaries only once. Existing de-duplication approaches, however, either rely on a shared OS to de-deduplicate binary objects, which provides unacceptably weak isolation; or exploit hypervisor-based de-duplication at the level of memory pages, which is blind to the semantics of the objects to be shared.

We describe *Object Reuse with Capabilities (ORC)*, which supports the fine-grained sharing of binary objects between tenants, while isolating tenants strongly through a small trusted computing base (TCB). ORC uses hardware support for memory capabilities to isolate tenants, which permits shared objects to be accessible to multiple tenants safely. Since ORC shares binary objects within a single address space through capabilities, it uses a new relocation type to create per-tenant state when loading shared objects. ORC supports the loading of objects by an untrusted guest, outside of its TCB, only verifying the safety of the loaded data. Our experiments show that ORC achieves a higher memory density with a lower overhead than hypervisor-based de-deduplication.

## 1 Introduction

In data centers, memory density determines how many applications can be deployed on machines with given memory amounts. Therefore, density is a critical cost factor, as memory contributes significantly to capital and operational expenses [3]. The challenge of achieving high memory density is expected to worsen as applications move to larger working set sizes [20, 36], and machines have more memory [10].

High memory density can be achieved by *de-duplicating* memory pages that have the same contents across a constellation of virtual machines (VMs), containers, and processes running on machines. This exploits that, in practice, the same OS is used across VMs, the same applications across containers, and the same libraries across processes [7, 31, 41, 61].

We observe that there is a trade-off between the efficiency of de-duplication and the level of isolation between tenants. Containers and processes achieve near-perfect memory density when they use a shared OS with binary loaders that explicitly identify de-duplication opportunities, e.g., through dynamic shared libraries [24, 25]. The high efficiency of de-duplication is due to the shared OS, which has visibility of memory at a *binary object level*. For security reasons, cloud environments, however, require stronger isolation between tenants, i.e., by using VMs without a shared OS.

In contrast, hypervisors implement strong isolation at the instruction set architecture (ISA) level, moving OS-level semantics to the guest OS. While this removes complexity from hypervisors, allowing them to provide strong isolation, it loses semantic information about how memory pages are used by VMs for object allocation. Memory de-duplication must thus occur at a *page level*: the hypervisor compares page contents blindly across VMs and performs expensive page table manipulations when de-duplicating, both of which result in performance and tail latency overheads [8, 39]. While hypervisors can accept de-duplication hints from VMs to reduce page scanning [1, 31, 40, 51], this does not eliminate overheads.

Our goal is to design a new cloud software stack that combines high memory density with low overhead, while providing strong isolation guarantees between tenants, relying only on a small trusted computing base (TCB).

We describe **Object Reuse with Capabilities (ORC)**, a new cloud software stack that allows de-duplication across tenants with strong isolation, a small TCB, and low overheads. ORC extends a binary program format (ELF [9]) to enable isolation domains to share binary objects, i.e., programs and libraries, by design. Object sharing is always explicit, thus avoiding the performance overheads of hypervisors with page de-duplication. For strong isolation, ORC only shares immutable and integrity-protected objects. To keep the TCB small, object loading is performed by the untrusted guest OS.

In more detail, the design and implementation of ORC combines the following novel features:



**(1) Object sharing with capabilities.** Current cloud stacks use page tables to control isolation and sharing, but page table manipulation is expensive: inter-VM sharing requires exits into the hypervisor to modify nested page tables [60]; de-duplication must temporarily downgrade page table entries, which can severely affect tail latencies [54].

Instead, ORC shares binary objects by design between isolation domains: it uses hardware support for *memory capabilities* [14, 16, 34, 64, 66] to place objects into *compartments*. Memory capabilities grant access to memory regions, can be copied between memory and registers, and are protected by hardware. They can be a building block for isolating cloud tenants with a small TCB by supporting an OS instance per compartment, as in today’s VMs [49].

With the help of capabilities, ORC isolates multiple compartments within a single address space, while sharing binary objects between compartments with virtually no overhead. ORC uses memory capabilities to isolate compartments within a single page table, safely and efficiently sharing objects in a controlled way.

**(2) Safe sharing of immutable objects.** Current binary formats, memory layouts, and loaders are designed for sharing across address spaces. After an object is loaded into memory, formats such as ELF [9] assume that global variables are mapped at fixed addresses relative to the code. While this is not an issue with per-process page tables, because an object’s global variables are mapped to different physical addresses in each process, ORC’s shared page table means that global per-process-and-object variables must be handled differently when sharing pages across compartments.

As a solution, ORC introduces a new type of variable relocation for *compartment-local storage* (CLS). ORC maintains absolute and code-relative references for code and read-only data, and the area for per-thread variables, i.e., thread-local storage (TLS). It also adds a new mechanism for per-process variables that replaces the traditional global variable references. This allows compartments to share immutable contents directly, i.e., code and read-only data, while still having per-process-and-thread state that is isolated across compartments. Under the hood, ORC allocates writable global variables in each compartment’s CLS, and loads objects to refer always to the executing compartment’s CLS (similar to TLS).

**(3) Untrusted loading of shared objects.** When objects are shared across isolation domains, loading is typically controlled by the TCB. The complexity of object loading bloats the TCB: it requires access to I/O devices, must load binary data into memory, and adjust memory contents to reflect load-time addresses, e.g., through relocations. Such functionality spans user-level, kernel-level and device driver code, and moving it into the TCB exposes a wide attack surface.

ORC avoids this issue by allowing untrusted compartments to handle most of the object loading (i.e., storage and file system I/O, data processing and copying, and adjustments

of memory contents). When an object is requested for the first time, the untrusted compartment manages its loading, and requests ORC to register an immutable and integrity-protected version of the newly-loaded object.

ORC verifies that the loaded object cannot be used to attack future compartments that reuse the same object, and makes it available to future load requests. The verification process is simple: it requires (i) scanning the memory contents of the registered object to calculate a hash, which ensures the object’s integrity in future load requests; and (ii) checking that any contained capabilities used by the object stay within the object’s memory and maintain immutability.

We evaluate ORC using a prototype implementation on the CHERI/Morello platform [42, 66] that includes a new compiler pass and loader support for CLS, a small privileged component that manages compartments and enforces the properties for secure sharing of binary objects, and a port of a library OS and C standard library that execute in each compartment.

We consider three workloads: (1) a cloud-based video transcoding micro-service, showing that ORC’s memory de-duplication is more resource-efficient compared to page-level de-duplication; (2) a latency-sensitive key/value store, demonstrating the lower impact of object-level de-duplication on tail latencies; and (3) an embedded database system, evaluating ORC’s decomposition cost into sharable compartments.

ORC has several limitations: unlike hypervisor-based de-duplication, ORC only de-duplicates read-only contents; it needs applications recompiled to use capability instructions and the new CLS; and it executes all compartments in one virtual address space, because capabilities use virtual addresses.

## 2 Increasing Memory Density in the Cloud

Memory density in cloud computing defines how efficiently the cloud provider is utilizing memory. Improving memory density is crucial for providers, because memory is often the main resource that determines how many tenants can be accommodated [33]. While providers want to exploit as many memory-sharing opportunities as possible, they must ensure that tenants and their workloads remain isolated.

We first discuss different approaches and their associated challenges for page-based isolation and memory sharing (§2.1). After that, we provide background on memory capabilities, which can act as an isolation mechanism without some of the drawbacks of page table-based isolation (§2.2).

### 2.1 Page-based memory sharing and isolation

Cloud tenants expect strong isolation for their applications from those of other tenants, while providers seek ways to minimize the total physical memory footprint by finding shareable memory. The two goals are at odds with each other: the

mechanisms that we have for efficient memory sharing are, in essence, reducing the level of isolation between tenants.

With today's isolation approaches, we must choose between containers [35, 38] and VMs [2, 30, 61], which in turn dictate how memory sharing can be done:

**(1) OS-managed memory sharing.** Container-based deployments rely on a shared OS for isolation. The OS provides user-space abstractions for sharing memory, and a loader can map the same binary object (e.g., dynamic library) across multiple processes. The OS has thus enough user-level information to de-duplicate object contents at load time, sharing memory across containers without extra runtime overhead.

Higher memory density in containers comes at the expense of isolation. Containers are not as strongly isolated as VMs, because they are managed by a shared OS kernel. Such a large, shared TCB is too complex to eliminate all vulnerabilities [11], which can be exploited by malicious containers to access information from other containers and tenants [12, 13].

**(2) Hypervisor-managed memory sharing.** VMs offer stronger isolation compared to containers: they expose a narrow virtualization interface at the level of the ISA, with a potentially small TCB (the hypervisor) that makes security vulnerabilities less likely [29, 57]. To share memory between VMs, typical hypervisors, such as ESX [61] and KVM [30], identify and eliminate redundant memory pages at runtime. Since the hypervisor lacks visibility into the semantics of user-space applications within each VM, it must periodically scan the memory of each VM to find pages with identical content, and remap these guest physical pages across VMs into the same host physical page to de-duplicate their contents.

A popular implementation of this approach is Linux *kernel same-page merging* (KSM) [1], which the KVM hypervisor leverages to eliminate duplicate memory pages across VMs. KSM periodically scans physical memory to find identical pages, and deduplicates them by mapping a single physical copy to multiple virtual locations. It also marks those pages as copy-on-write (COW), which triggers the re-duplication before a modification on shared page contents. Therefore, each VM instance can safely operate on its own copy, without affecting the memory contents of other VMs.

KSM uses red-black trees to search for memory pages with identical content. For efficiency, it utilizes two trees: (1) a *stable tree* that contains already-shared pages; and (2) an *unstable tree* that represents pages not shared but scanned previously. During the scanning process of a memory page, KSM first searches for a match in the stable tree. If the page is found, the redundant copy is eliminated through merging. If there is no match in the stable tree, KSM checks whether the page has been modified since the last scanning round by comparing hashes. If the page has not been modified, it is considered a suitable candidate for searching in the unstable tree. If the page is found in the unstable tree, merging occurs, and the shared page is inserted into the stable tree. Otherwise,

it is inserted into the unstable tree as a scanned page. The unstable tree is also reinitialized after each scanning round.

Despite the advantages of hypervisor-based isolation, its memory de-duplication mechanism has several drawbacks: (1) blindly scanning page contents and manipulating their permissions comes with an overhead on average performance and tail latencies [8, 39, 54]; (2) hypervisors lack the visibility of an OS to application-level load-time semantics, and therefore must rely on page scans; (3) while the use of larger pages in the cloud improves memory performance [27, 47], it can reduce memory density by making memory de-duplication less frequent; and (4) the use of COW semantics on de-duplicated pages has been shown to be vulnerable to timing side-channel attacks across VMs [26, 45, 58, 59, 67, 69].

## 2.2 Isolation with memory capabilities

As described above, using paging for both translation and protection introduces performance challenges in traditional virtualized environments due to its management granularity. In contrast, *memory capabilities* offer an alternative memory protection mechanism that is more flexible, robust, and efficient to manage. At the same time, memory capabilities can co-exist with the use of paging for translation [4, 6, 18, 19, 64].

Memory capabilities replace integer-type pointers with protected capabilities. Unlike regular pointers, capabilities provide information to enforce accesses within a given address range and access type. They can thus be used to partition a single address space into multiple, isolated regions, allowing the use of a single page table across isolation domains.

**Memory capabilities.** The Capability Hardware Enhanced RISC Instructions (CHERI) architecture [62] provides a modern implementation of memory capabilities. It introduces new instructions, registers, and other hardware primitives to support capabilities. CHERI enforces three properties: (1) *provenance validity* ensures that a capability cannot be created from an arbitrary sequence of bytes, but can only be derived from another capability; (2) *capability integrity* guarantees that capabilities in memory cannot be modified. One-bit *validity tags* are used to distinguish them from other data for protection; and (3) *monotonicity* ensures that a capability's permissions, including its bounds, cannot be expanded but only reduced.

CHERI can thus replace all pointers in an application with capabilities to enforce precise bounds and permissions on each memory access. This is known as the *pure-capability* (*pure-cap*) mode. Pure-cap enables spatial memory safety and fine-grained memory sharing, but requires ABI changes and other source code changes, e.g., to pointer-integer casts. CHERI also supports a *hybrid* mode, in which code is permitted to use legacy, capability-unaware instructions. In this mode, all control flow and memory operations are checked against a pair of special registers that contain *program-counter* and *default data capabilities*, thus restricting the code and data accesses performed by capability-unaware instructions.

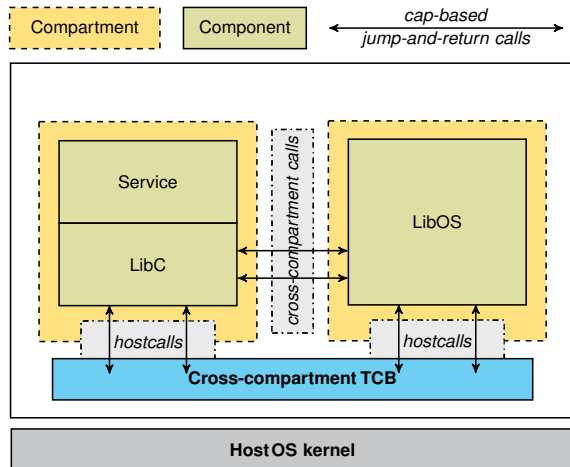


Fig. 1: Using capabilities to create compartments

Code can use the capability-aware `CInvoke` instruction to perform function calls between isolated memory domains, carrying the necessary capability arguments across domains.

**Capability-based compartmentalization.** Capabilities offer a good mechanism to isolate software components, and their flexibility and efficiency can eliminate the overheads of page-based sharing. CAP-VM [49] describes a new capability-based compartmentalization mechanism: each capability compartment (cVM) has its own separate address sub-range within a shared address space, and executes programs in CHERI’s hybrid mode. cVMs are managed by a shared TCB component called the *Intravisor*, similar to a VM hypervisor. The *Intravisor* is a host process that starts cVMs as one or more host threads within its address space, using the default capabilities in hybrid mode to isolate cVMs. Each cVM has its own library OS instance to support private namespaces and program execution environments.

Fig. 1 shows how multiple components can be placed in capability compartments, potentially sharing access to objects across compartments. An *Intravisor*, or some other cross-compartment TCB, can give each compartment the capabilities needed to jump into/call code in other compartments, allowing the compartments to share capabilities to the same object. The figure also shows how capabilities can be used to request *Intravisor* operations (*hostcalls* at the bottom).

After compiling software components using CHERI’s pure-cap mode, however, it is not possible to use capability-based compartments to share objects efficiently, because: (1) the *Intravisor* has no explicit information about the extent and sharing properties of objects; and (2) existing storage formats and memory layouts of pure-cap binary objects assume that each compartment has its own page table.

In particular, ELF [9] assumes that global variables are reachable through constant addresses relative to code locations. If we use a per-process (or compartment) page table,

we can physically share non-writable pages across processes, while having separate contents for writable pages. This is no longer the case if we use a single page table, which is the only way to avoid page table management overheads.

## 2.3 Efficiency and security considerations

The goal of this paper is to provide high memory density in cloud environments. To achieve this goal, our solution must fulfill the following requirements:

(1) *Strong isolation with a minimal TCB:* Memory sharing is needed for density, but it should not undermine isolation between tenants. We must thus reduce the attack surface by providing a small TCB with a narrow interface to manage isolation and sharing for density.

(2) *Low performance overhead:* The sharing mechanism should not incur high overheads in terms of CPU cycles, and it should not prevent the system from performing other optimizations, such as using large pages to reduce TLB misses.

(3) *High sharing precision:* The sharing mechanism should support arbitrary object sizes and have visibility into the intended object sharing semantics. An ideal solution should not miss opportunities to share, nor unintentionally share memory that soon diverges into different contents. This can be a problem with KSM, because it blindly de-duplicates pages solely based on content and access frequencies.

Note that sharing memory can be used as an unintended *side channel* across compartments. This is an intrinsic trade-off between memory density and isolation that all cloud providers face, regardless of the employed mechanism. Given the importance of side channels, there are proposals to avoid or mitigate them at both hardware and software levels [23, 44].

The sharing of binary objects in this work is limited to side channels on accesses to (i) code and (ii) read-only data only. Since the user controls which binary objects to share and when, they can decide on a suitable policy for trading off between memory efficiency and side-channel resistance. We leave the exploration of such policies to future work.

## 3 Design of ORC

We exploit the capabilities provided by the CHERI architecture [62] to implement both software compartmentalization and binary object sharing. ORC shares binary objects *explicitly*, making the use of physical memory denser without the performance overheads of de-duplication.

Fig. 2 shows an example with two applications (app1 and app2), which use multiple object binaries that are identical across VMs, including the OS kernel (database, libC and kernel). Fig. 2a shows a baseline system in which each application is deployed in a VM for maximum isolation. In this case, the hypervisor incurs overheads of memory de-duplication.

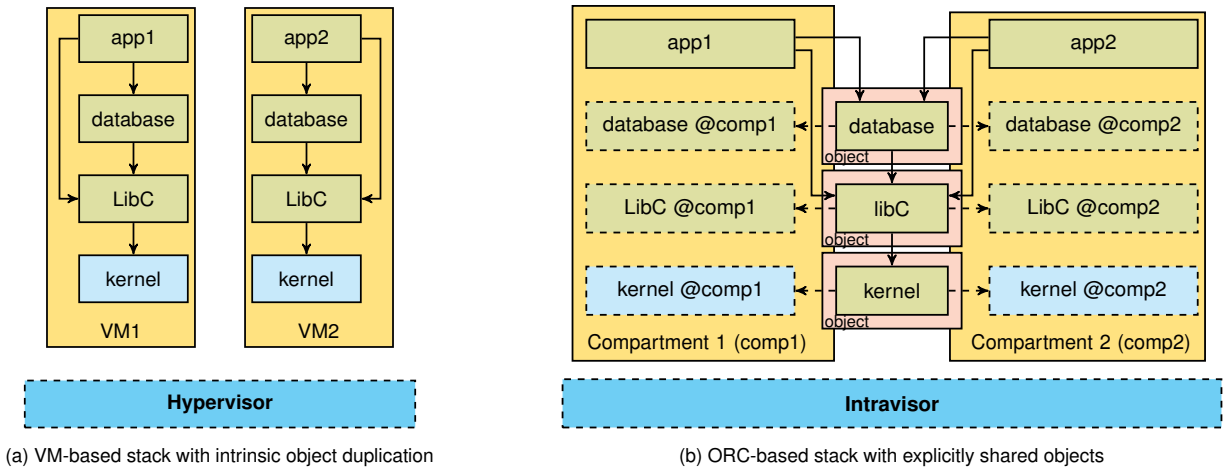


Fig. 2: Comparison between VMs and ORC compartments with explicit object sharing (Dotted lines show compartment-local variables.)

In contrast, Fig. 2b shows the approach taken by ORC. Programs are isolated into *compartments* (shown as light yellow boxes), which contain all needed objects (app1, app2, database and libC) as well as their own OS kernel instance (kernel). ORC compartments are deployed using a shared page table, and obtain access to non-overlapping addresses using *memory capabilities*. Both the page table and capabilities are controlled by the TCB, shown as *Intravisor*.

Compartments are strongly isolated: each has its own OS instance, and they are restricted to access non-overlapping memory address ranges. Sharing objects across compartments is supported through capabilities, which provide access to the object’s contents (light red boxes with objects database, libC and kernel). If possible, the ORC program loader requests capabilities from the Intravisor for an object that has already been loaded by another compartment. Otherwise, the compartment loads the object itself and registers it with the Intravisor, allowing future compartments to reuse it.

To make object sharing across compartments safe, the Intravisor must ensure that a shared object cannot be modified after registration. This, of course, implies that the registering compartment cannot change the object after registering it, but also that shared objects cannot contain writable state. We indicate this with the dotted lines in Fig. 2b: each shared object is recompiled to have per-compartment instances of any writable state. We refer to this as *compartment-local storage (CLS)*.

As a result, objects can be efficiently shared across domains, while retaining strong isolation down to the level of separate OS instances. In addition, sharing is part of the cloud software stack, ensuring that the memory density benefits do not come at the cost of reduced performance.

### 3.1 Architecture overview

Fig. 3 shows the high-level architecture of ORC. Programs execute within a compartment (shown as yellow boxes) and

have statically and dynamically-linked objects, as usual.

- 1 All potentially shareable objects, including the main program binary, must be compiled with our ORC-specific extensions. These extensions move all the writable state of an object into the CLS, i.e., all global writable variables, by extending the binary storage format and the loader (see below). The figure shows an example with three global variables, a constant, a thread-local, and a writable variable (var0, var1, and var2, respectively). Of the three variables, only var2 is moved to the CLS, because thread-local variables are already stored in a per-thread data structure, the TLS [17].

Note that none of these elements are part of the TCB – compartments can still use private copies of objects without the ORC extensions, and only references to global writable variables are changed to the CLS. Heap allocations are supported as usual, resulting in private compartment allocations.

Each compartment has its own, untrusted loader (ld.so in Linux). The compartment itself therefore loads the required objects by reading them from storage and parses their contents according to the binary format (e.g., ELF). 2 When the loader finds an ORC shareable object, it allocates the necessary memory to load the object and prepares it for execution.

- 3 After loading, the compartment calls the Intravisor’s `orc_register()` operation to register the loaded object for future use. The compartment passes the capabilities to where the object’s contents are loaded, and a list of variable references in the object’s code. The Intravisor then copies the object to a new location controlled by itself, computes a hash of its contents, resolves the variable references to the new load address, and registers the object’s hash and allocation capabilities for future use. At this point, all compartments, including the registering one, proceed in the knowledge that the shareable object is available in the Intravisor.

- 4 When a shareable object is registered in the Intravisor, the compartment requests it via `orc_request()`, passing it the expected object content hash. If the hash matches that of



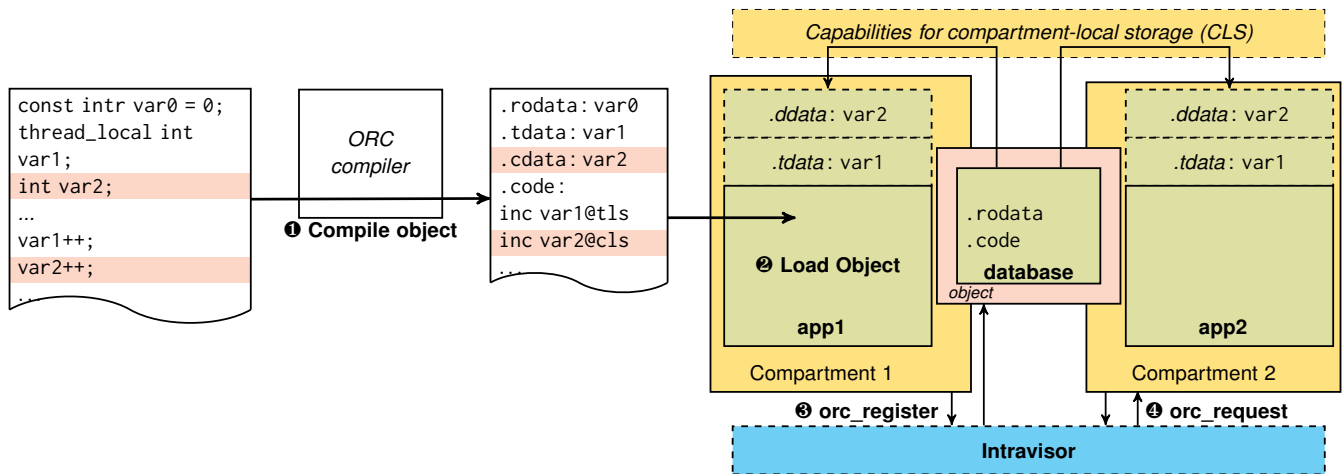


Fig. 3: Architecture of the ORC stack (Circled numbers identify operations referenced in the text.)

a registered object by `orc_register()` above, the Intravisor releases the capabilities for that object. The loader then proceeds by allocating the memory for the CLS variables, and subsequently loaded objects can reference this one.

Note that the main program itself can be an ORC shareable object. In that case, after loading it with `orc_request()`, the whole application is ready for execution.

### 3.2 Compiler support and binary format

To support shared ORC objects, the compiler extends the binary format with CLS variables. With ORC enabled, the compiler adds a flag to identify the object as ORC-enabled, and moves writable global variables to the CLS.

For CLS, the compiler replaces each reference to a writable global variable with a new relocation type. Such relocations are resolved at load time to point to the per-container instance of that variable (see below), similar to how thread-local variables are moved to the TLS at load time.

Fig. 3 shows this process on the left-hand side. Code, constant variables, such as `var0`, and thread-local variables, such as `var1`, are handled as usual by the compiler: they are placed in the `.code`, `.rodata` and `.tdata` sections of the ELF object, respectively, generated with standard relocations. Writable global variables (`var2`) are placed in a new `.cdata` section, and references identified with the new `@cls` relocation.

### 3.3 Secure object loading and reuse

The compartment loader brings the object's file contents into memory, and handles all relocations that are independent of the load address. It then calls the Intravisor's `orc_register()` operation by passing the capabilities that delimit the memory regions in which the object was loaded and a description of the yet-unprocessed relocations.

To ensure that the object contents cannot be changed once shared, the Intravisor allocates new memory in capabilities  $C$ , copies the object contents into them, and checks that the object contains no capabilities pointing outside the  $C$  allocations to avoid malicious use of `orc_register()`. At this point, the Intravisor computes a hash  $H$  of the object contents, resolves the remaining relocations that depend on the information of the secure load location, and registers the object's hash and a non-writable version of the allocation capabilities,  $H$  and  $C$ , respectively. The CLS relocations are replaced with a value that points to a per-compartment memory address that holds all CLS variables of that object (see §4.2).

When a compartment calls `orc_request()`, it passes the hash  $H$ . If an object with hash  $H$  exists in the Intravisor, i.e., it was previously registered with `orc_register()`, the Intravisor returns the capabilities for it, which are ready to use by the calling compartment.

The object hash  $H$  is computed by the Intravisor before any location-dependent relocations, and so it is also known by the requesting compartment. If an object with hash  $H$  exists in the Intravisor, we know that its integrity and isolation are ensured. The Intravisor does not ensure object correctness beyond relocation resolution, which should be handled through other means, e.g., attestation checks as part of the software supply chain, for which hash  $H$  can be helpful.

### 3.4 Discussion

With ORC, components are shared by design via capabilities. Component sharing by design could also be implemented by other systems, e.g., traditional hypervisors with MMU mappings, but end-to-end performance would vary due to the different hardware mechanisms for enforcing isolation. Capabilities have further benefits e.g., their ability to provide spatial memory protection within components.

We also note that ORC only de-duplicates read-only mem-

ory contents, while traditional hypervisors also de-duplicate writable pages. This is not necessarily an issue: ORC de-duplicates application instances in less time than hypervisor-based approaches, which makes it better suited for short-lived instances, as often found in cloud environments. Our evaluation results also show that de-duplicating small amounts of writable memory leads to little practical benefit.

Finally, ORC requires the recompilation of applications, because applications must use capability instructions for code and data access and ORC's compiler pass for the required CLS functionality. ORC also executes all compartments in a single virtual address space, but we expect this to not be a problem: virtual addresses are an abundant resource, and it is possible to use large blocks to make allocations scalable and resistant to side channels e.g., via core-local caches.

## 4 Implementation

We implement ORC on the Morello platform [42], a development board from Arm that supports the CHERI capability extensions. In this section, we describe how we: build ORC compartments by extending CAP-VMs [49] with our own library OS to maximize object sharing; add CLS support through a new LLVM compiler pass; and implement the necessary logic to load shared objects securely.

Both the Intravisor and the host OS are implemented as hybrid capability code using the existing CAP-VM and CheriBSD [22] projects. We extend the Intravisor to support pure-cap compartments, i.e., using the pure-cap CHERI ABI, and the program loading operations, which adds 530 and 240 lines of C and assembly code, respectively.

For our evaluation, we also port the SQLite database [56], FFmpeg [21] with the libav libraries, and Redis [48] to support the pure-cap CHERI ABI and ORC. In total, the porting requires approximately 350 lines of code. Note that, besides adding the system functionality specific to ORC, the main effort went into porting code to the pure-cap model.

### 4.1 Library OS and standard C library

To increase object sharing across compartments, we make the library OS and low-level C library support a pure-cap build, as no such software components exist with the necessary functionality. We implement our own pure-cap library OS kernel, which is based on Unikraft [32] and CubicleOS [50], from which we use 40 system calls and 9,061 lines of code. We extend it with support for the CHERI ABI and add a capability-aware memory allocator.

We also use a pure-cap version of the C library for our evaluation applications. It is based on musl libc [43], whose pure-cap support is maintained by Arm. Our fork has 494 functions and 19,717 lines of code. We modify it to introduce a capability-aware memory allocator based on `d1malloc`, and a few extra changes for compatibility with our library OS.

### 4.2 Compiler support and CLS

We implement our ORC compiler support as an LLVM pass. It replaces all references to global, writable, non-TLS variables with a call to function `__cls_get_addr()`, which returns a compartment-local version of that variable. Internally, `__cls_get_addr()` is implemented using regular capability-aware instructions (`cgetaddr`, `cincffset`, `csetlen`, etc.). The function retrieves the address of the variable from the input capability and makes it relative to the beginning of the data section. After that, it applies the relative offset to the capability that points to the shadow data section. Finally, it creates the replaced capability by limiting its size to match that of the original capability.

The `__cls_get_addr()` function works similarly to how TLS is supported, i.e., through `__tls_get_addr()` in ELF [17]. It takes the shared object identifier (assigned at load time) and the variable offset within the CLS (assigned at compile time), and returns a capability that grants access to the calling compartment's copy of that variable.

For ease of implementation, the compiler pass inlines `__cls_get_addr()` into the generated code – a production implementation should use a separate CLS relocation type – and it uses a TLS variable to point to the compartment's CLS buffer for that object. This means that the loader (see below) only needs to implement TLS variables, accessed through the `tp` register in Arm, to support both TLS and CLS.

### 4.3 Secure object loader

To simplify application deployment, we implement a loader that takes deployment configurations, i.e., a list of binary paths to load into memory. The program deployment logic loads binaries into the target memory regions, resolves relocations, and generates all capabilities needed in the PLT and GOT of a pure-cap program [63].

The deployment configuration also identifies shareable objects and provides their hash, so that they can be reused if previously loaded by `orc_register()` and `orc_request()`.

### 4.4 Discussion

Our prototype implementation showcases ORC's core ideas, but it has shortcomings:

*Performance.* The CLS implementation uses TLS variables for simplicity. This results in new capabilities fetched from TLS and adjusted to the corresponding variable on each call to `__cls_get_addr()` (except for reuse optimizations in the compiler). In a future version, we would pre-calculate the per-variable capability at dynamic link time, so that no new capabilities are created at runtime by `__cls_get_addr()`.

*Compatibility.* Since we use a compiler pass, we cannot support pre-initialized variable references on other data structures,

e.g., a statically-initialized array entry pointing to a CLS variable address. In our library OS (Unikraft), we found only a single place where this was necessary. Adding compiler support for new CLS relocations would solve this problem by adjusting data structure addresses at dynamic link time.

Our `__cls_get_addr()` implementation assumes a per-compartment CLS. We plan to support per-process CLS, allowing for multiple processes within the same compartment.

## 5 Evaluation

We ask the following questions when evaluating ORC: (1) how efficient is ORC compared to KSM? (2) what is the impact of ORC on tail latency compared to KSM; and (3) what is the execution and compilation overhead of ORC, as a function of the degree of binary object sharing?

### 5.1 Experimental setup

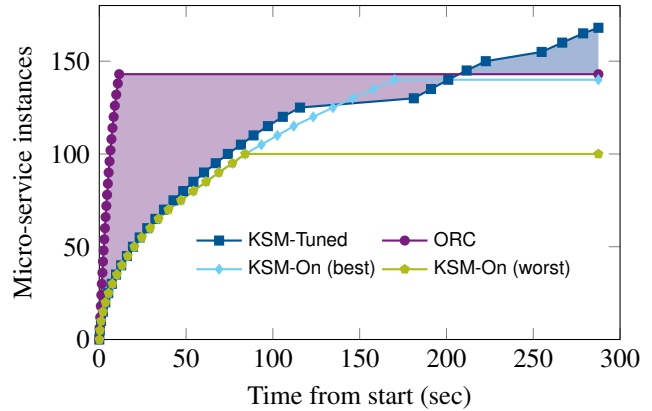
**Workloads.** We evaluate a real-time video transcoding micro-service that scales out to a large number of clients. The micro-service uses *FFmpeg* [21] to perform transcoding. A single *FFmpeg* instance consumes a fraction of the CPU and memory resources on the machine, allowing multiple instances to be run. By de-duplicating memory, we can support more *FFmpeg* instances and thus more concurrent clients. We compare the deployment of the micro-service using ORC to one that uses Linux KSM as a baseline for memory de-duplication.

We also consider other workloads to evaluate specific characteristics of ORC: (1) we use Redis [48] with the *memtier* benchmark [37] to understand the impact of ORC and KSM on request tail latencies; and (2) we use the SQLite database system [56] and its *speedtest1* benchmark [55] to compare the performance of different object sharing scenarios.

**Testbed.** We deploy ORC on a Morello board [42], which has an Armv8-A CPU with hardware support for CHERI [66]. The board has 4 CPU cores running at 2.5 GHz, with 16 GB of DDR4 memory (64 KB L1, 1 MB L2, and 1 MB L3 caches).

The experiments compare two OSs: (1) Ubuntu 22.04.1 LTS with Linux v5.15.0, which only runs native arm64 binaries with no CHERI support; and (2) Hybrid CheriBSD version 14 (release/22.05p1) [22]. The Linux OS is used to measure KSM, and can also run the entire ORC stack without isolation guarantees (i.e., disabling our compiler pass and eliminating capability management instructions in the Intravisor and loader). The CheriBSD OS is used to run ORC with all its isolation guarantees, as described in this paper. All ORC results use CheriBSD unless stated otherwise.

Note that the same source code executes on all compartments, so that we can compare ORC and KSM despite the different compiler options and underlying OS support.



**Fig. 4: Throughput over time when de-duplication of the video transcoding micro-service happens** (The shaded area indicates the difference between ORC and KSM-Tuned, which is the best-performing KSM configuration.)

### 5.2 Efficiency and performance overhead

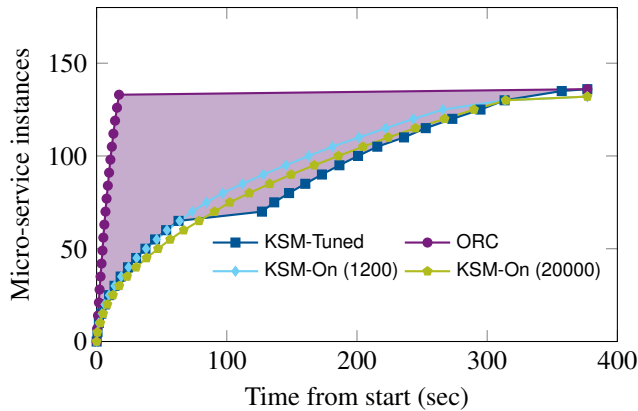
We now evaluate the trade-off between memory de-duplication efficiency and application performance overhead when comparing ORC and KSM. We deploy the video transcoding micro-service, which increases application throughput with higher memory density – i.e., it increases the number of processed frames by increasing the number of deployed transcoder instances.

The experiment deploys new transcoder instances until it reaches one of two limits: (i) memory limit – when the instances consume all available physical memory, but there are still spare CPU resources to support more instances; and (ii) CPU limit – when at least one of the instances can no longer transcode at real-time due to a lack of CPU resources.

Each micro-service instance contains *FFmpeg*’s main program and libraries, our library OS, and the standard C library (see §4.1). It occupies around 111 MB of memory (11 MB in binaries and read-only data, shareable by ORC, and 100 MB of heap). The transcoded video has a resolution and frames-per-second configuration such that, without de-duplication, the experiment reaches the memory limit after 127 instances; optimal de-duplication can run more instances: it eventually reaches the CPU limit after 180 instances.

We deploy multiple KSM configurations with different de-duplication and overhead trade-offs:

KSM-Tuned is the default policy of the *ksmtuned* daemon [70]. Every 60 secs, it checks the share of free memory, and starts KSM if it is below 20%. It also adjusts KSM parameters: the number of scanned pages in each iteration (*pages\_to\_scan*) is increased gradually when the de-duplication rate is too low. KSM-On is a hand-tuned policy that achieves good memory efficiency in the shortest time for our workload, but it consumes significant CPU resources. With this setting, KSM operates constantly and uses 20,000 *pages\_to\_scan*.



**Fig. 5: Throughput over time when de-duplication of the video transcoding micro-service happens using larger binaries** (The shaded area indicates the difference between ORC and KSM-Tuned.)

**Micro-service instances.** Fig. 4 shows the number of active micro-service instances over time, with ORC and the various KSM configurations. Given the performance of one instance, there are sufficient CPU resources for 180 instances, but only enough memory for 127 instances without de-duplication.

The plot shows that KSM-Tuned first reaches the memory limit (127 instances) after 119 secs, and it does not de-duplicate memory until 58 secs later. It then reaches the memory limit again at around 227 secs, and, after 23 secs, it is able to de-duplicate enough memory to deploy 180 instances.

In contrast, ORC is designed to optimize for density almost instantaneously: it creates 142 instances in just 11 secs, which is a 20 $\times$  speedup over KSM-Tuned for the same number of instances. Note that the 142 instances deployed by ORC correspond to 11% more than the 127 instances without de-duplication. This is expected, because code and read-only data occupy only 11% of the memory of each instance (see above). In contrast, KSM can de-duplicate additional pages by also considering writable memory.

ORC thus deploys and de-duplicates instances much faster than KSM-Tuned, which gives it an advantage over KSM on the aggregate performance over time, expressed as the total number of processed frames (highlighted by the shaded areas in Fig. 4). Although KSM-Tuned deploys more instances than ORC within 300 secs, at this point, ORC has processed 15%–35% more frames than the various KSM configurations. To outperform ORC, KSM-Tuned would require a total execution time of 441 secs – an extra 141 secs after reaching 180 instances. None of the other policies outperform ORC.

KSM-Tuned limits its de-duplication speed, as it tries to minimize CPU overheads – it operates only under memory pressure (80%), and at a limited memory scanning rate (`pages_to_scan`). This negatively impacts environments in which processes are frequently created and destroyed, so we also evaluate the case in which KSM maximizes de-duplication rate with KSM-On. Since KSM is probabilistic in

**Tab. 1: Impact of memory de-duplication on Redis tail latency**

	set requests		get requests	
	p50	p99	p50	p99
Linux	0.5 ms	9.7 ms	0.5 ms	9.3 ms
Linux+KSM	0.5 ms	20.9 ms	0.5 ms	20.8 ms
CheriBSD	2.1 ms	3.7 ms	2.2 ms	3.6 ms
CheriBSD+CC	2.1 ms	4.2 ms	2.1 ms	4.3 ms
CheriBSD+CC+ORC	2.1 ms	4.9 ms	2.1 ms	4.8 ms

nature, we show the best and worst results of twenty different runs of the same KSM-On experiment.

KSM-On (best) deploys more instances than KSM-Tuned within the first 170 secs, but instances are created more slowly and peak at just 140 (98.6% of ORC), because KSM is constantly consuming more CPU resources. KSM-On (worst) creates instances at the same rate as KSM-On (best), until its heuristics stop at only 100 instances (70% of ORC).

*Conclusions:* The results show that instance creation and de-duplication in ORC are substantially more efficient than in the baseline system with KSM. Although KSM-Tuned de-duplicates more (writable) memory, given enough time, ORC retains an advantage with shorter-lived application instances, which are prominent in the cloud. In addition, more aggressive de-duplication with KSM-On is not viable, because higher de-duplication rates are hidden by higher CPU overheads, resulting in a 30% throughput overhead compared to ORC.

**Large binary instances.** We now evaluate how a larger amount of shareable memory affects ORC and KSM, given that language runtimes and programming frameworks can easily consume hundreds of megabytes.<sup>1</sup> To this end, we run the same experiment after manually injecting an additional 100 MB of code into the FFmpeg binary, resulting in 53% of shareable code and read-only data contents on each instance.

Fig. 5 shows the results for this experiment, which supports 68 and 136 instances without and with perfect de-duplication, respectively. In this case, the ability to de-duplicate writable data in KSM has no long-term advantage over ORC, because KSM has further CPU overheads that prevent spawning additional instances; ORC reaches 136 instances in 18 secs, while it takes KSM-Tuned 377 secs to reach the same maximum. In this case, we also report the best results for KSM-On with two fixed values for `pages_to_scan` (1,200 and 20,000), which reach 132 instances 52 secs earlier than the default policy.

*Conclusions:* With a larger proportion of directly shareable contents, ORC reaches optimal de-duplication effectiveness at a 20 $\times$  faster rate than any of the KSM configurations.



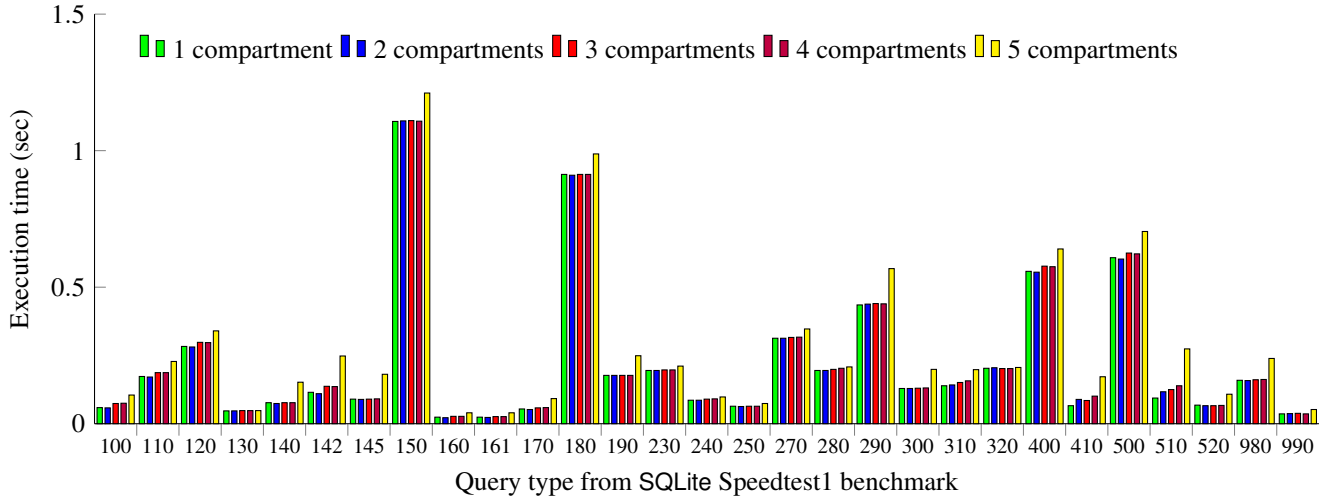


Fig. 6: Query execution times for different compartments using ORC (SQLite)

### 5.3 Impact on service tail latency

Next, we investigate the impact of ORC and KSM on the tail latencies in a typical cloud service. We observe that KSM consumes up to an entire CPU core when active, and it issues TLB shutdowns when scanning and de-duplicating pages.

We spawn 4 instances of the Redis key-value store service [48] and use the *memtier* benchmark [37] as a workload: it pre-fills each instance with 2.5G GB of data and then executes a 1:10 set/get request workload, using 4 threads with 4 concurrent connections for each instance.

Tab. 1 shows the results of five deployments: (1) Linux acts as our baseline (arm64 binaries without using CHERI or KSM); (2) Linux+KSM adds memory de-duplication with an always-on KSM; (3) CheriBSD is our baseline with a CHERI-capable host OS but without ORC or enabling capabilities when compiling Redis; (4) CheriBSD+CC uses ORC to compartmentalize Redis but disables de-duplication; and (5) CheriBSD+CC+ORC uses ORC for de-duplication.

We run both Linux and CheriBSD to decouple the impact of KSM and ORC from the intrinsic differences between the two OSs. CheriBSD shows worse throughput and tail latencies than Linux on all operations, which can be attributed to the different device driver and network stack implementations.

Linux+KSM matches the 50<sup>th</sup> percentile (p50) latencies of Linux, but the CPU overheads and TLB shutdowns due to KSM more than double the 99<sup>th</sup> percentile (p99) latencies. In contrast, ORC has a small impact on tail latencies: support for compartmentalization alone (CheriBSD+CC, i.e., compiling the program in pure-cap mode and crossing isolation boundaries results in a 13% and 19% increase in p99 latencies

<sup>1</sup>For example, the MongoDB database system uses 104 MB; the PyTorch machine learning stack with CUDA uses over 200 MB; a Python-based data science pipeline with TensorFlow uses over 300 MB.

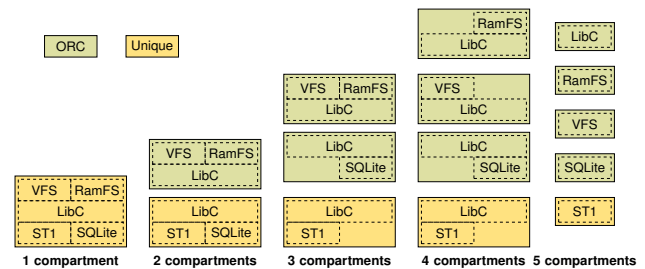


Fig. 7: Decomposition into capability compartments (SQLite)

for set and get operations, respectively. Fully enabling ORC (CheriBSD+CC+ORC) leads to an 17% and 12% increase in p99 set/get latencies, respectively, which can be attributed to the overheads of our current CLS implementation.

*Conclusions:* The page table management and memory hashing overheads of KSM’s page de-duplication lead to a more than 2× increase in p99 latencies; the explicit sharing of binary objects in ORC reduces these overheads to 12%–17%.

### 5.4 Cost of isolation

We also investigate how increasing the number of shareable binary objects in an application affects performance. We control both the overheads introduced by the compiler pass, and those of capability-based crossings between binary objects.

To measure the cost of isolation, we execute a single compartment with the *speedtest1* benchmark and its embedded SQLite instance, and incrementally make some of its components separate binary objects (see Fig. 7). We start with one compartment that contains all components in a single binary object. Since CLS support is unnecessary here, we disable the compiler pass. We then incrementally build further compo-

nents into separate shareable objects (2 to 5 compartments), compiled with CLS support. The *VFS* and *RamFS* components correspond to internal components of the library OS; *STI* is the benchmark binary.

Fig. 6 shows the execution times of the benchmark for different SQLite query types, varying compartment numbers. We observe that all configurations, except for 5 compartments, have only a minor performance overhead. The average difference between one compartment (everything monolithic; no CLS support) and 4 compartments (all system components but LibC are shareable objects) is 10% (median of 3%). Even when isolating all components into separate shareable objects (5 compartments), which adds many cross-compartment calls, the average slowdown is 53% (median of 39%).

*Conclusions:* We draw two conclusions: (1) the overhead of supporting CLS is small, especially when compared to the performance cost of cross-object calls; and (2) while the cost of cross-object calls exists, it is sufficiently small that it becomes possible to share across multiple fine-grained objects.

## 6 Related Work

**Page-level memory sharing.** Modern hypervisors support dynamic page sharing among VMs. VMware ESX [61] pioneered inter-VM page sharing without guest OS support by periodically scanning physical pages and transparently discovering pages with identical contents using their hash values. KSM [1] also periodically scans physical pages but uses a balanced tree to find duplicated pages (see §2.1). Dynamic page sharing by hypervisors, however, results in an inflexible sharing granularity due to the lack of OS semantics, unpredictable latency spikes due to runtime scans, and vulnerabilities to side-channel attacks due to copy-on-write semantics.

Hypervisors can also perform page sharing on disk reads. Disco [5] intervenes in DMA to support copy-on-write shared disks and copy-less NFS shares among VMs, allowing page sharing without runtime scanning. Satori [41] uses similar sharing-aware block devices that enable copy-on-write sharing as well as content-based sharing through enlightenment (para-virtualization). Sharing in these systems is still page-based, and copy-on-write issues remain.

VM introspection (VMI) or graybox approaches can be used to improve the efficiency of de-duplication by extracting semantic information from in-VM memory data. Sindelar et al. [53] used VMI techniques to identify memory pages belonging to free memory pools in Windows and Linux without making kernel version-specific assumptions. They are treated as zero pages to improve de-duplication and VM migration efficiency. Singleton [52] uses KSM page information to de-duplicate pages in the guest page cache by dropping them from the host page cache. VMI, however, cannot always obtain semantic information reliably without cooperation from the guest, and these techniques are still page-based.

Overall, ORC has the advantage over page-level sharing in that it allows for reliable, flexible, and efficient sharing that leverages semantic information about objects.

Efficient inter-VM page sharing techniques can be applied to optimize inter-server VM placement for cloud-wide memory density. *Memory buddies* [65] aggregate memory fingerprint information into a centralized control plane to determine VM placements for increased sharing and to optimize VM placement dynamically with live migration. Sindelar et al. [53] show that inter-VM sharing largely occurs hierarchically, and they propose a tree structure to manage sharing. ORC leverages semantic knowledge about shared memory objects and could be applied to improve memory density in the cloud through optimal VM placement.

**Mixed-granularity sharing.** Sharing at a granularity finer than pages can help increase memory density, as many pages are found to be nearly identical with only some differences. Gupta et al. [28] propose a *difference engine* as an extension to Xen [2], which supports sharing at the sub-page level in addition to page level. The difference engine stores patches against reference pages for similar but not identical pages, and compresses pages that are unique but accessed infrequently. Several studies on VM live migration also leverage sub-page granularity for differentiation, compression and write detection [15, 46, 71]. Although such proposals could increase memory density, unlike ORC, they do not reason about what should be shared across tenants.

Prior work on improving scanning efficiency groups pages based on access characteristics and uses a granularity finer than pages. CMD [8] identifies page access characteristics by measuring the distribution of writes per subpage using dedicated hardware, in addition to the address and number of writes to the page. UKSM [68] proposes adaptive partial hashing, which hashes only a portion of the page and gradually changes its size. These approaches allow for fine-grained sharing and reduce the cost of hashing, but still lack semantic information, making sharing opportunities non-deterministic.

In modern cloud environments, it has become important to leverage large page sizes that minimize the overhead of TLB misses. The opportunity for page-level memory sharing decreases with the page size. SmartMD [27] splits large cold pages with high repetition rates for de-duplication, while re-consolidating small hot pages for improved access performance. GLUE [47] attempts to maintain large-page performance in regions that are broken into small pages for de-duplication. It extends the hardware to perform speculative large-page translation using normal-sized TLBs. While these approaches leverage finer granularities than a page, ORC has an advantage in its ability to share objects at byte granularity.

## 7 Conclusions

We have described ORC, a new memory de-duplication approach that improves the memory density of cloud environments using capability-protected compartments. Our motivation was to create a practical, capability-based cloud stack, in which tenants can enjoy strong isolation and cloud providers benefit from more efficient use of memory resources.

Unlike conventional hypervisors, which blindly scan memory for identical pages, ORC takes advantage of a semantic separation into sharable and non-sharable objects. Therefore, it is not subject to the performance overheads of existing runtime methods. Due to its use of capabilities, ORC can share objects more precisely at a word granularity (i.e., spatial precision), while avoiding unintentional sharing of runtime objects (i.e., temporal precision). The loading of objects is done via a narrow interface with a small TCB, providing strong isolation.

**Source code availability.** The source code of ORC and our evaluated sample applications can be downloaded from <https://github.com/llds/intravisor>.

**Acknowledgements.** This work was funded by the Technology Innovation Institute (TII) through its Secure Systems Research Center (SSRC), and the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme (UKRI grant EP/V000365 "CloudCAP"). This work was also supported by JSPS KAKENHI grant number 18KK0310 and JST CREST grant number JPMJCR22M3, Japan. We thank our shepherd, Malte Schwarzkopf, and the anonymous reviewers for their helpful feedback and comments.

## References

- [1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium 2009*, pages 19–28, 2009.
- [2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, pages 164–177. ACM, 2003.
- [3] Luiz André Barroso, Jimmy Clidaras, and Urs Hlzl. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2nd edition, 2013.
- [4] Viktors Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture, ISCA '80*, pages 245–252, New York, NY, USA, May 1980. Association for Computing Machinery.
- [5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997. Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI*, pages 319–327, New York, NY, USA, 1994. Association for Computing Machinery.
- [7] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 244–249. IEEE, 2011.
- [8] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based Memory Deduplication through Page Access Characteristics. *ACM SIGPLAN Notices*, 49(7):65–76, 2014.
- [9] TIS Committee et al. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2, 1995.
- [10] Intel Corporation. 5-level paging and 5-level EPT. Technical report, Intel Corporation, May 2017. Revision 1.1.
- [11] Domenico Cotroneo, Roberto Natella, and Roberto Pietrantuono. Predicting aging-related bugs using software complexity metrics. *Performance Evaluation*, 70(3):163–178, 2013. Publisher: Elsevier.
- [12] CVE-2013-6441. Available from MITRE, CVE-ID CVE-2013-6441, December 2013.
- [13] CVE-2021-21284. Available from MITRE, CVE-ID CVE-2021-21284, December 2021.
- [14] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.
- [15] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live Gang Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC '11*, page 135–146, 2011.
- [16] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *ACM*

- SIGOPS Operating Systems Review*, 42(2):103–114, 2008.
- [17] Ulrich Drepper. *ELF Handling For Thread-Local Storage*, August 2013. Version 0.21.
- [18] DM England. Capability concept mechanism and structure in System 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 63–82, 1974.
- [19] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.
- [20] Wei Fan and Albert Bifet. Mining big data: Current status, and forecast to the future. *ACM SIGKDD Wxplo-rations Newsletter*, 14(2), 2013.
- [21] FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. <https://ffmpeg.org>. 2022.
- [22] FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. <https://github.com/CTSRD-CHERI/cheribsd>. Last accessed: June 1, 2022.
- [23] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.
- [24] Robert A Gingell, Meng Lee, Xuong T Dang, and Mary S Weeks. Shared libraries in SunOS. *AUUGN*, 8(5):112, 1987.
- [25] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.
- [26] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, volume 9326, pages 108–122. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.
- [27] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 733–744, 2017.
- [28] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 309–322, December 2008.
- [29] Gernot Heiser. The seL4 microkernel – an introduction, June 2020. White paper. The seL4 Foundation, Revision 1.2 of 2020-06-10.
- [30] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.
- [31] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Efficient memory sharing in the Xen virtual machine monitor. *Aalborg University*, pages 1–86, 2006.
- [32] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuve, Sharan Santhanam, Alexander Jung, Gauthier Gain, Cyril Soldani, Costin Lupu, Ștefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.
- [33] Sajib Kundu, Raju Rangaswami, Ming Zhao, Ajay Gulati, and Kaushik Dutta. Revenue Driven Resource Allocation for Virtualized Data Centers. In *2015 IEEE International Conference on Autonomic Computing*, pages 197–206, July 2015.
- [34] Lanfranco Lopriore. Capability based tagged architectures. *IEEE transactions on computers*, 33(09):786–803, 1984.
- [35] Linux containers. <https://linuxcontainers.org>. Last accessed: June 1, 2022.
- [36] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Intl. Symp. on Microarchitecture (MICRO)*, 2019.
- [37] NoSQL Redis and Memcache traffic generation and benchmarking tool. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark). Last accessed: Dec 13, 2022.
- [38] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.
- [39] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, 2013.
- [40] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, 2013.



- [41] Grzegorz Milos, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *USENIX Annual technical conference*, 2009.
- [42] Arm Morello Program. <https://www.arm.com/architecture/cpu/morello>. 2022.
- [43] musl libc. <https://musl.libc.org>. Last accessed: June 1, 2022.
- [44] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [45] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–8, November 2011. ISSN: 2374-9628.
- [46] Yosuke Ozawa and Takahiro Shinagawa. Exploiting Sub-page Write Protection for VM Live Migration. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 484–490, 2021.
- [47] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 1–12, Waikiki Hawaii, December 2015. ACM.
- [48] Redis is an in-memory database that persists on disk. <https://github.com/redis/redis>. Last accessed: June 1, 2022.
- [49] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 597–612, Carlsbad, CA, July 2022. USENIX Association.
- [50] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, pages 575–587. ACM, 2021.
- [51] Martin Schwidetzky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Proceedings of the Linux Symposium*, volume 2, pages 313–330. Linux Symposium Incorporation Ottawa, 2006.
- [52] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing - HPDC ’12*, page 15, Delft, The Netherlands, 2012. ACM Press.
- [53] Michael Sindelar, Ramesh K. Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA ’11*, page 367, San Jose, California, USA, 2011. ACM Press.
- [54] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: a near-memory content-aware page-merging architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 302–314, 2017.
- [55] Speedtest1 benchmark. <http://www.sqlite.org/src/finfo?name=test/speedtest1.c>. Last accessed: Dec 13, 2022.
- [56] SQLite. <https://www.sqlite.org>. 2022.
- [57] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the Fifth European Conference on Computer Systems*, EuroSys ’10, pages 209–222. ACM, 2010.
- [58] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Software side channel attack on memory deduplication. In *ACM Symposium on Operating Systems Principles (SOSP 2011), Poster session*, 2011.
- [59] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Implementation of a Memory Disclosure Attack on Memory Deduplication of Virtual Machines. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E96.A(1):215–224, 2013.
- [60] A Virtualization. Secure virtual machine architecture reference manual. *AMD Publication*, 33047, 2005.
- [61] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2003)*, December 2003.
- [62] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical report, University of Cambridge, Computer Laboratory, 2019.

- [63] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. *CHERI C/C++ Programming Guide*. Technical report, University of Cambridge, Computer Laboratory, 2020.
- [64] Maurice Vincent Wilkes and Roger Michael Needham. *The Cambridge CAP computer and its operating system*. *Operating and Programming System Series*, 1979.
- [65] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review*, 43(3):27–36, 2009. Publisher: ACM New York, NY, USA.
- [66] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.
- [67] Lei Xia and Peter A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date - VTDC '12*, page 11, Delft, The Netherlands, 2012. ACM Press.
- [68] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 325–339, February 2018.
- [69] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, Budapest, Hungary, June 2013. IEEE.
- [70] Bernd Zeimetz. ksmtuned. <https://github.com/bzed/debian-ksmtuned>. 2022.
- [71] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, pages 88–96, 2010.



# Global Capacity Management With Flux

Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang

Meta Platforms

## Abstract

Customers of both private and public clouds must wrestle with the problem of *regionalization*: how should service capacity be apportioned across a large number of geo-distributed datacenter regions? This problem is further complicated by the complex service dependency graphs that arise from microservice architectures, as well as capacity availability and hardware mix that can vary greatly by region.

Historically, regionalization has been solved through a slow-moving and manual process, whereby owners of large services directly negotiate capacity allocation and distribution with the cloud provider. However, as both service and cloud footprints continue to grow, these manual processes are becoming untenable, and often result in excessive labor for all parties involved, as well as suboptimal outcomes.

At Meta, we have built a system called *Flux* to automate capacity regionalization, transitioning it from a bottoms-up, manual process, to a top-down, automated one. Flux employs RPC tracing to identify service capacity models, and uses these to compute an optimal joint capacity and traffic distribution plan that spans thousands of services across tens of products, and involves millions of servers. These plans are orchestrated by a system that safely and efficiently rebalances service capacity and product traffic across tens of regions on a continuous basis.

## 1 Introduction

Meta's private cloud consists of millions of machines and hosts products serving billions of users. It must provide the products with a growing, geographically distributed capacity footprint, so that they can scale and remain fault tolerant while their usage grows and new features are introduced.

Our products employ large microservice architectures [29] comprising thousands of services, globally deployed in tens of datacenter regions. Most of these services are *shared* by multiple products, and hence the sizing of one service needs to consider the demands of all products. Moreover, the services are *interdependent*. It is not uncommon that a service calls tens of other services and the depth of the call graph can

go beyond 10 levels. As a result, capacity distributions for services must be managed in concert: the growth of one service may cause the growth of tens more, which in turn places further capacity demands on their downstream services, and so on. Thus, it is a daunting task to manage capacity at scale, as service operators must answer questions such as: How to size my service? How and where do I provision capacity for organic growth, or to enable a new feature? Are downstream services correctly provisioned for the demand generated by my service?

To avoid the above complexities getting out of control when planning capacity jointly across tens of regions, service owners often prefer the simplicity of reasoning about their capacity on a per-region basis: a service responds to demand increases by requesting that new capacity be delivered to the regions where the service is already running.

However, this local optimization leads to many issues:

- Services in mature regions cannot grow as those regions have no available space or power to add capacity.
- Hardware utilization becomes imbalanced as some regions may provide more capacity than others.
- Hardware ordering is overly constrained by specific services requiring specific hardware to be placed in specific regions.
- Capacity imbalances lead to excess disaster-readiness buffers as we must have enough buffers for the potential loss of the single *largest* region.

Before Flux, these issues were solved by lengthy negotiations between service owners and cloud providers. For example, in order for service A to grow in region X, the service owners might negotiate to trade the service's excess capacity in region Y for service B's capacity in region X. This laborious process does not scale well with the number of regions and services, and leads to suboptimal capacity allocation. Moreover, even after capacity negotiations, rebalancing services and traffic across regions in order to match capacity supply is still a daunting task, requiring coordination across many services and traffic distribution systems.



Finally, this region-centric capacity management process leads to tight coupling between specific products and specific regions. The *capacity mix* (i.e., the ratio of different hardware types) of any given region often reflects the products that have historically been deployed to that region. As a result, the capacity mixes of our regions have already diverged significantly, further exacerbating the problem as this regional heterogeneity limits the flexibility of moving services across regions to rebalance demands and supplies.

**Global capacity management.** Our key insight to solving these issues is to elevate capacity management to a *global* problem, decoupling global service placement from global hardware placement. This paper describes our global capacity placement system, called Flux, which runs continuously on weeklong timescales, redistributing service capacity and product traffic to best utilize our global capacity footprint.

Flux utilizes RPC tracing to identify a model that predicts service capacity demands given a traffic mix for our products. This model is then used to formulate a mixed-integer-programming (MIP) problem that jointly distributes service capacity and product traffic across all of Meta’s regions. Flux’s service placement distributes service growth capacity, rebalances existing service capacity to meet infrastructure goals, and manages projected regional capacity deficits such as those caused by regional hardware refresh.

A *capacity orchestrator* works with our existing autoscaling and traffic management systems to safely and efficiently execute global capacity redistribution plans. The orchestrator also allows human-in-the-loop operations as needed by escalating low-quality estimates to be vetted by human operators, or delegate orchestration for services that have not yet onboarded to our autoscaling systems.

Flux has been running in production at Meta for 2.5 years, continually allocating service capacity quotas and performing cross-region shifts of service placements and traffic for our largest products. Currently, Flux covers about 50% of the servers in our private cloud that consists of millions of servers supporting online, batch, and AI training & inference workloads. We expect Flux to cover more than 90% of our capacity as we increase adoption. Flux has also enabled dramatic simplification of our hardware planning process, as it allows us to plan for capacity globally, while gradually homogenizing our current heterogeneous regional hardware mixes.

Besides Flux’s usage in our private cloud, the ideas presented in this paper can potentially be adapted to public cloud settings as well. Public cloud providers also similarly negotiate with their large customers directly to match capacity demands with supplies. This sometimes entails providing capacity outside of the customer’s preferred regions, which in turn may require customers to relocate their workloads.

**Contributions.** We make the following contributions:

- To the best of our knowledge, this paper is the first to conduct a comprehensive study of global capacity manage-

ment and global service placement, which are important issues for public and private cloud providers. We hope that by sharing our firsthand experiences, we can help the research community better understand this important problem and the constraints involved in solving it.

- We propose *global* capacity contracts, whereby service owners only need to reason about their global capacity demands, leaving it for Flux to optimally *regionalize* service capacity and product traffic distributions. By contrast, cloud providers still mostly operate in a mode where large services require specific hardware to be placed in specific regions, and traffic distribution is not integrated with capacity management.
- We use RPC tracing to build a service-capacity regionalization model, which calculates a service’s regional capacity distribution as a function of the traffic mix for different products. We are not aware of any prior work that attempts to use models to regionalize service capacity. Moreover, although RPC tracing has been used for debugging and performance modeling, we are not aware of any prior work that uses it for capacity modeling, not to mention doing it at our scale and in production.
- We formulate a MIP problem to optimally distribute service capacity under constraints of capacity supply as well as service and infrastructure objectives. Despite the widespread use of MIP, our approach is novel in its application to joint capacity and traffic regionalization, a problem that has not been considered before. We also use load-test-induced nonlinear models to complement MIP-based linear models, improving modeling accuracy.
- We describe our *capacity orchestrator* which integrates across autoscaling and traffic management systems to safely implement capacity and traffic redistribution plans. Automating joint service placement and traffic redistribution at our scale is highly risky and may negatively impact site reliability. To our knowledge, this has not been attempted before.
- Finally, the effectiveness and robustness of Flux is demonstrated by the fact that we use it every quarter to assign hundreds of thousands of new machines to services.

## 2 Background

In this section, we provide background on our datacenters, workloads, capacity management practices, and the capacity management challenges that are addressed by Flux.

### 2.1 Datacenter Regions

Meta operates 10s of datacenter regions, each comprising multiple datacenter buildings in the same local geography, typically located on a single campus.

Our existing infrastructure abstractions generally operate at the level of regions. For example, our cluster management

systems [37, 45] manage all machines in a region as a single pool. Services are expected to be oblivious to the placement of their tasks within a region, and they may be spread across multiple datacenter, network segments, or power domains. Our regional infrastructure abstractions are supported by a network fabric that provides sufficient regional cross-sectional bandwidth for all but a few workloads

## 2.2 Traffic Management

We maintain a global network of small edge datacenters that are connected to our backbone network. User traffic (e.g., those from apps or web browsers) are terminated in these edge datacenters. Requests from clients connected to our edge datacenters are forwarded to front-end servers in one of our geo-distributed regions. A traffic distribution system [18] manages the distribution of requests from edge datacenters to our large datacenter regions, typically by considering a combination of factors including front-end utilization and geographic proximity to the end-user.

## 2.3 Service Workloads

Traffic enters a region through a front-end web service, typically an HTTP reverse proxy that routes the request to an appropriate application server based on the request URI. The application server implements some business logic, and typically makes RPC calls to tens to hundreds of services, which in turn fan out to yet more services. While some of these services implement functionality used by just a single product, most are shared across many products. Thus, our services are highly interdependent, and we cannot partition our services into product-specific silos.

Figure 1 illustrates the complexity in our request serving paths, with a large fanout and deep call depth. The complexity of service interdependencies [29] motivated us to use precise RPC tracing to attribute resource consumption when designing Flux, rather than using indirect methods such as statistical analysis [2] or heuristics [3, 38, 43].

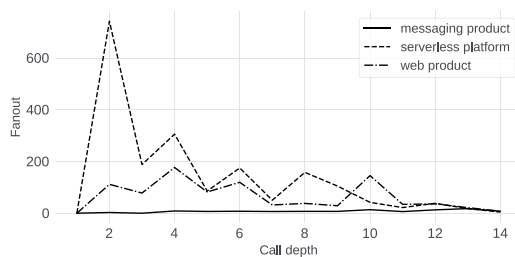


Figure 1: **Service RPC fanout.** An example of how to read the curves: when a request for the web product reaches the call depth of 10, it fans out to call 158 services on average.

## 2.4 Service Capacity Management

Meta’s capacity management systems provide quotas in the form of regional reservations [37], which provide strong guarantees of regional capacity availability and sub-regional failure tolerance. Thus, the various hardware buffers required to reliably operate services within a region are encapsulated by the regional capacity management systems, and hidden from higher-level capacity management systems like Flux. The number of service replicas in a region is usually determined by our autoscaling systems which combine service capacity models with demand forecasts and disaster scenario simulations to ensure the job is sized correctly.

Most RPCs occur within the same region due to strict latency requirements imposed by applications. Additionally, our complex service dependency graphs often contain critical paths with tens of hops, which can quickly amplify cross-region RPC latencies. Finally, by hosting both the caller and callee in the same region, we can limit cross-region dependencies and improve disaster readiness [31, 52].

Figure 2 illustrates regional caller-callee affinity by showing the latency distribution of RPC calls across thousands of compute services, denominated by total capacity. The inflection point at around 25ms represents calls going cross-region; observe that  $\approx 80\%$  of capacity is reached by in-region requests.

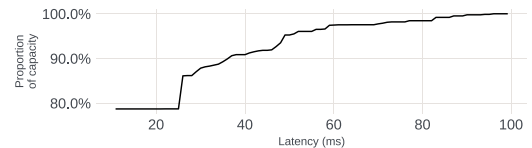


Figure 2: **Cumulative distribution of capacity by RPC latency.** Note the y-axis begins at 80%.

All of our products are located in multiple regions to improve disaster readiness, access a large capacity pool, and achieve wide geographic distribution. However, the distribution of service capacity across regions is uneven. This is due to the organic growth of both service capacity demand and regional capacity supply. Service capacity demand responds to new features and world events, while regional capacity supply depends on factors such as power availability and datacenter construction timelines. Additionally, geographic skew in product usage exacerbates the issue as we try to place service capacity close to end users.

Over time, this has led to a negative feedback loop, wherein services prefer to grow proportionally to their existing regional footprints, causing regional hardware mixes to reflect these historical workloads. This in turn makes it difficult to move these workloads to other regions, causing services to continue preferring the regions in which they are already deployed to receive capacity growth. We can see this specialization reflected in the regional hardware mix, as shown

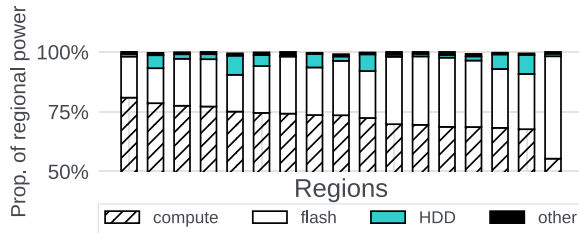


Figure 3: **Regional hardware type distribution.** A histogram of the compute, flash, HDD, and other hardware deployed across a subset of our regions. Note that the y axis starts at 50%. The right-most region is dominated by flash, because it is a small region that recently underwent a large retrofit, leaving a lot of database workloads in place.

in Figure 3. For instance, the percentage of servers of the compute type ranges from 55% to 80%.

Regions also vary in their power headroom, i.e., the difference between used and available power. Observe in Figure 4 that six regions have little available power, implying that if service deployments in those regions need to grow, they must expand their capacity footprint in other regions as there is little additional power to support additional racks. This is similar to the situation in public clouds where users cannot acquire new capacity in a given region [10–12, 21, 22, 47].

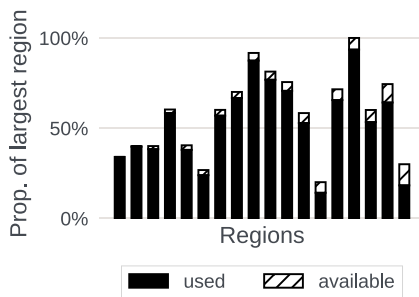


Figure 4: **Power headroom per region for capacity growth.** The y-axis is power, normalized to our largest region.

## 2.5 Capacity Management Challenges

Service capacity management presents significant challenges for both service owners and infrastructure operators. While service owners wish to grow freely where they already are deployed, infrastructure operators must reconcile these wishes with constraints associated with operating physical infrastructure, as well as goals around efficient fleet operations.

These problems are amplified as regions reach maturity—when there is no longer additional power available to allocate to new racks—and infrastructure owners cannot accommodate service capacity growth without shrinking the footprint of other services. The complex dependency graphs between services in a typical online product make this a more challenging problem still.

We refer to this challenge as the *service regionalization problem*: How can we optimally allocate capacity to a set of services across multiple regions? Additionally, how should product traffic be appropriately distributed based on this allocation? Finally, considering that cluster and capacity management systems usually operate at the regional level, how can we effectively rebalance services according to the regionalization plan?

## 3 Design and Implementation

Our global capacity management system, *Flux*, continually rebalances a large number of interdependent services across regions in response to demand changes (e.g., product growth) and supply changes (e.g., hardware refreshes). By decoupling the management of capacity demand and supply, Flux enables service owners to focus on their global capacity demand without considering regional needs, and allows cloud providers to evolve each region’s capacity supply independently.

### 3.1 Overview of Flux’s Workflow

As illustrated in Figure 5, Flux solves the regionalization problem through the following workflow.

**Product-to-service capacity attribution via RPC tracing.** Flux uses RPC request tracing [30, 40] to construct a regional *baseline* ① that attributes each service’s peak capacity footprint to the products that are served directly or indirectly by the service. This baseline is used to construct a service placement model ② that determines how service capacity should be distributed given a product traffic distribution.

**Joint regionalization of service capacity and product traffic.** Using inputs from our budgeting systems, Flux then creates a capacity *target* for each service, which specifies the amount of global capacity that is needed for each service. These service targets are jointly regionalized with product traffic by formulating an assignment problem that is solved using mixed-integer programming (MIP) ③. The result of this stage is a *placement plan* that redistributes service capacity and product traffic across regions. The optimization problem also encodes a number of infrastructure objectives, such as minimizing the total amount of disaster-readiness buffer required to operate services safely.

**Global capacity orchestration.** Flux introduces a global capacity orchestrator, responsible for executing placement plans safely and efficiently ④. The orchestrator drives automation through several capacity and traffic management systems, and also supports human-in-the-loop operations to handle exceptions or uncertainties in execution.

The overall Flux workflow runs continuously in weeklong cycles to rebalance service capacity across regions according to changing hardware supplies and service demands. It measures the current state of service and hardware placement, computes a desired state, and then executes a plan to move

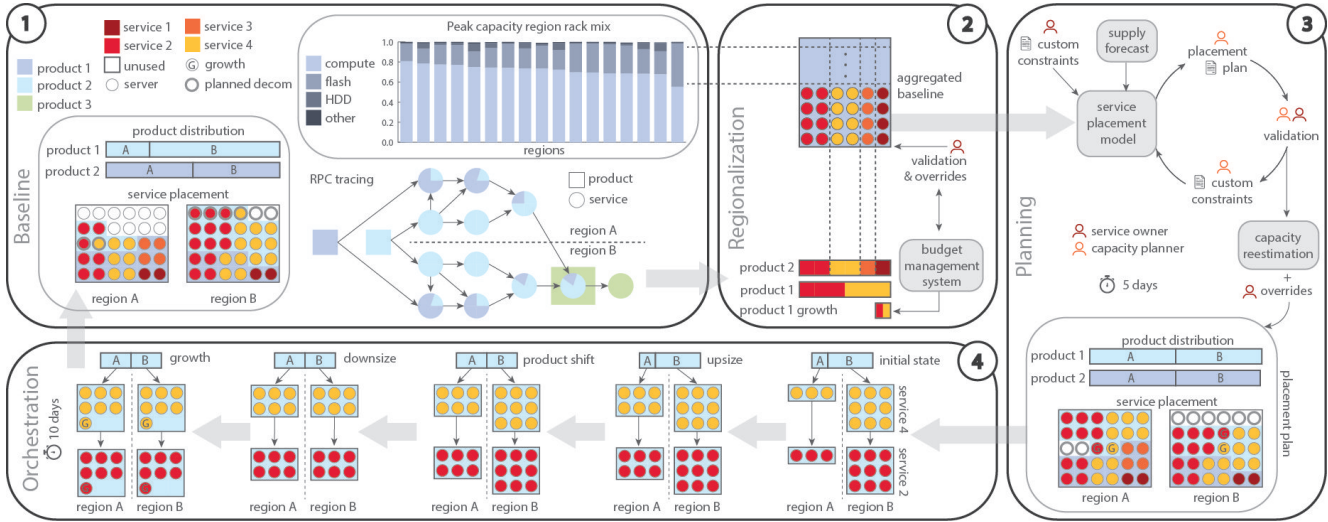


Figure 5: Flux's end-to-end workflow.

towards the desired state. It tolerates imperfect execution of the plan by repeating this self-correcting reconciliation loop.

Next, we describe the details of the steps above.

### 3.2 Service Modeling

The goal of service capacity modeling is to determine the optimal distribution of a service's capacity across regions in concert with product traffic. This is challenging because many of our services are shared by multiple products, and each product may invoke different call paths within the service. This can result in significant differences in the cost per request depending on the product being served (see §2.4).

#### 3.2.1 Baseline

Flux defines a *baseline* for each region, attributing portions of each service's peak capacity footprint to different products. This baseline is created by combining two other baselines:

**Capacity usage baseline.** We run profilers on every server in our fleet to produce a dataset that attributes resource usage to specific services. Profiles are sampled every minute, and we process this dataset to identify the daily peak time window<sup>1</sup> and peak resource usage per service and per region, covering different resource types such as CPU and SSD.

**Demand baseline.** Flux uses sampled RPC traces to reconstruct the call graphs for requests that are handled by each service. We identify a set of *product gateways* that act as traffic entry points for each product. Importantly, the traffic destined for these gateways is globally and independently routable, and is usually managed by Meta's shared traffic management systems [18]. Each sampled RPC call is attributed

<sup>1</sup>Demand spikes due to new product launches or special events such as New Year's Eve are handled separately. During daily off-peak periods, many of our services are automatically scaled down to donate unused capacity to our elastic capacity pools, which are used to run preemptible services.

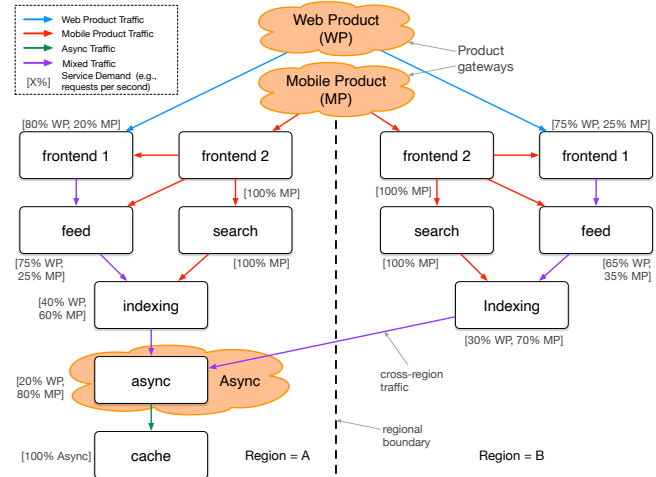


Figure 6: **Demand attribution.** Attribution of capacity usage to three Products: Web Product, Mobile Product, and Async. For the indexing service in Region A, the annotation “[40% WP, 60% MP]” means that 40% and 60% of the indexing service's capacity consumption is attributed to the Web Product and Mobile Product, respectively. Note that while demand attribution is relative, the capacity usage baseline is defined in terms of absolute capacity.

to the product handled by the nearest upstream gateway in the call path. The demand attribution process is illustrated in Figure 6. Sampled traces are aggregated to compose the demand attribution dataset for each service, dividing a service's total demand among the set of products served by it. Traces collect multiple demand metrics, including call counts as well as CPU instructions. The next section discusses how we select the demand metric that minimizes the overall model error.



### 3.2.2 Modeling

Flux’s service models predict the amount of service capacity required in a region to serve that region’s product traffic mix. We assume that each product’s traffic is fungible across regions, and thus that capacity requirements for serving a fixed portion of the product’s traffic is also the same across regions.

This suggests a model where capacity for service  $s$  in region  $r$ ,  $c_{s,r}$ , is given by a linear combination of the capacity contributions from each product:

$$c_{s,r} = \sum_{p \in P} (\alpha_{s,p} * \rho_{p,r}) + \delta_{s,r}, \quad (1)$$

where  $\alpha_{s,p}$  is the amount of capacity for service  $s$  attributed to product  $p$ ;  $\rho_{p,r}$  is the *presence*, or the proportion of global traffic for product  $p$  assigned to region  $r$ ; and  $\delta_{s,r}$  is the model residual for service  $s$  in region  $r$ .

Modeling residuals may be due to an existing capacity imbalance, or due to nonlinear effects not captured by the model. We allow service owners to be involved in managing the treatment of residuals during planning.

The baseline includes multiple demand metrics. In the modeling step, we select the metric that minimizes modeling error. For example, many large services have per-request costs that vary significantly across different products (because they invoke different internal code paths), and are well-modeled using CPU instructions as a demand metric. On the other hand, some services perform very little computation for each request. Therefore, call counts are a more appropriate demand metric for these services due to the CPU overhead of tracing.

### 3.3 Joint Capacity & Traffic Regionalization

Flux computes joint service capacity and traffic regionalization plans by formulating an assignment problem that is solved by a MIP solver. This section provides the intuition behind the problem formulation.

The formulation, detailed in appendix A, is an optimization problem that jointly assigns capacity for each service in each region and product traffic in each region, corresponding to  $c_{s,r}$  and  $\rho_{p,r}$  from equation 1. The assignments are subject to a set of constraints imputed from the service placement model described in §3.2, which determines the service capacity mix required to serve each product.

Initial capacity and traffic assignment are given by the baseline. The capacity residual ( $\delta_{s,r}$  in equation 1) is interpreted as capacity that is unexplained by the model, and is thus excluded from reassignment, unless directed otherwise by the service owner. We provide an analysis of capacity residuals in §6.3.

**Baseline adjustments.** Flux adjusts the existing *baselines* to match planned capacity and product distribution changes. These planned changes are encoded as events and maintained in a *capacity ledger*. Flux commits its plans to the ledger

along with other capacity planning systems. Thus, Flux can overlap planning and execution, as we can adjust the baseline to account for planned changes between the plan generation time and the start of its execution cycle.

**Regional capacity pools.** Flux divides each region’s capacity into a shared pool per hardware type. Our capacity reservation system, RAS [37], provides these pools as a regional abstraction that we build upon, and lets us treat the capacity in each pool fungibly.

Each hardware type is assigned a capacity measure that represents the common bottleneck for that hardware type. For example, the generic compute pool is denominated by a normalized CPU throughput measure, while our SSD hardware is generally I/O bound. The capacity measure is normalized across all generations of the same hardware type. Some services can run on multiple hardware types: we encode this knowledge through a set of fungibility rules that establish a service’s conversation ratios between different hardware types.

**Service capacity demand.** The *global* capacity demand for each service is computed by querying our budgeting systems which mandate service capacity budgets in terms of a normalized cost measure. Flux converts this normalized budget to a hardware-type specific capacity demand using a conversion ratio specific to the service and hardware type.

**Service placement model.** Flux imputes placement constraints from the service capacity model. Specifically, for each service, the model determines a *lower bound* of service capacity assigned to a region as a function of the product traffic mix to that region (see §3.2.2). The product traffic assignments are also optimization variables, and hence the service and traffic placement is jointly optimized. The baseline model residual (see Equation 1) is codified as an explicit term in the formulation to offset capacity imbalances that exist at baseline. Flux gives service owners the choice to reduce the model residual, which is often used to correct baseline capacity imbalances; see §6.5 for an example.

**Optimization constraints.** The MIP assignment problem constrains (1) the capacity assignment in each region to be no more than its available supply; and (2) the global capacity demand for each service to be met. The latter constraint is a soft constraint, which allows us to prioritize capacity fulfillment among services if necessary. Flux prioritizes *baseline* capacity footprint (i.e., the capacity present when the baseline was measured) over *growth* capacity (i.e., additional capacity granted by the budget systems), to ensure that already granted service capacity is not taken away.

**Optimization objectives.** The MIP assignment problem includes several infrastructure objectives that help us manage our global capacity footprint more effectively. First, a balancing objective spreads service capacity evenly across regions, which reduces the amount of buffer capacity needed

for disaster readiness. Second, unused capacity is also distributed evenly based on region size, making extra capacity available to account for discrepancies or defects in Flux’s placement. Third, a stability objective minimizes the amount of capacity reassignment in each placement cycle, simplifying the placement plans and reducing infrastructure churn.

**Timesteps.** Regionalization is simultaneously computed for multiple future timesteps in increments of future execution cycles. This serves three purposes. First, multi-timestep plans can incorporate large changes in supply or demand ahead of time, allowing for a plan that anticipates these changes with small, individually feasible adjustments over multiple timesteps. Second, we can set stability objectives that span multiple timesteps to prevent undue oscillation in placement plans. Finally, this multi-timestep approach prevents the solver from optimizing a short-term solution at the expense of long-term negative impacts.

While Flux computes plans for multiple timesteps, we only execute the plan for the next timestep. Flux runs in a self-correcting reconciliation loop as the baseline can change between executions for a number of reasons, including: (1) execution may have deviated from the placement plan; (2) supply and demand forecasts may change in the interim; (3) manual capacity operations may affect the baseline; (4) service code changes or new services may affect the service models.

The formulation is detailed in appendix A.

### 3.4 Execution Planning

Flux derives an *execution plan* from the joint service and traffic placement plan. The plan is a directed acyclic graph (DAG) of service capacity assignments and product traffic assignments. The plan ensures that services are always sufficiently provisioned for the traffic during the transition stage.

Flux provides this guarantee through a three-phase plan. First, all upsizes are executed, i.e., each service is sized to the maximum of its baseline size and its target placement size. Second, all product traffic is reassigned. Third, downsizes are executed by sizing each service to its target size.

The advantage of this approach is its simplicity and the ability to execute it quickly by parallelizing actions in each phase. A disadvantage is that it requires temporarily overprovisioning services, which takes up capacity that could be used by other services. To address this limitation, we explored using more sophisticated multi-step plans. However, we found that these plans were both complex to execute and difficult to explain to service owners. As a result, we decided to continue using the simple approach.

### 3.5 Orchestration

The execution plan is fulfilled by Flux’s capacity orchestrator which: (1) executes capacity and traffic assignments in the

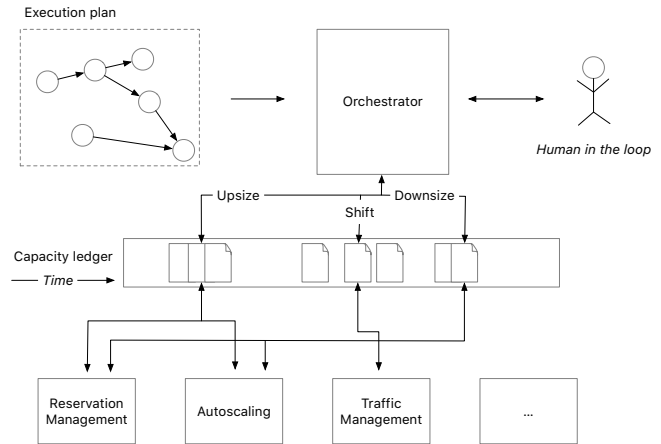


Figure 7: End-to-end orchestration workflow.

correct dependency order; (2) continuously monitors product-level and service-level metrics to ensure that they remain healthy; (3) delegates exceptions and actions to human operators as needed; and finally (4) performs load tests to validate the placement.

Figure 7 illustrates the orchestration workflow. A *capacity ledger* stores a timeline of *capacity events*. These events are timestamped, and each reflect a proposed capacity related change. Capacity and traffic management systems query the ledger for future events, but only execute them once they are marked by the orchestrator as *active*. After execution, the same systems store their status (success or failure) back into the ledger.

The ledger acts as a central repository of all anticipated capacity changes, and allows multiple systems to simultaneously propose and coordinate changes, while decoupling *capacity planning systems* from *capacity management systems*. While Flux is the primary writer to the ledger, we sometimes write manual events to make temporary capacity changes in support of product launches or experimentation.

The ledger provides three important properties. First, we can compose events from different writers, so that the underlying management systems can consider the combined effect of a set of events. Second, by providing events ahead of time, we accommodate services that require a long lead time to provision capacity and scale. For example, our caching systems need a significant warm-up period before newly provisioned capacity can handle production traffic. By providing future events, the control plane gives the management systems enough time to ensure that services are ready for future traffic shifts. Third, the ledger serves as an authoritative forecast of future capacity changes, and is used by Flux to incorporate future and ongoing events during planning. This allows Flux to overlap planning with execution, and to compose well with other capacity planning systems.

The orchestrator ensures that events are executed in dependency order by verifying that all antecedent events have

completed before marking an event as active. RF delegates any exceptions to human operators through its UI. The UI is also used when (1) the service owner has configured the service to require validation before execution, or (2) the capacity estimates for a service are considered low confidence in the modeling stage and require operator validation. By providing this progressive path towards full automation, Flux offers transparency and explainability, and allows service owners to gain comfort with the system.

Finally, before downsizing service capacity, the orchestrator initiates product load tests [52] to validate that the sizing is correct and that the site as a whole remains disaster ready.

### 3.6 Stateful Services

Flux integrates with Shard Manager [32] to handle stateful services within our platform. Shard Manager is responsible for managing most of our stateful workloads. Shard Manager continuously queries the capacity ledger for relevant capacity events, and builds new replicas after upsize capacity has been provided by Flux. This is done by migrating or replicating data from other regions. Shard Manager then acknowledges the capacity event, allowing Flux to safely proceed with execution. After the demand shift, Shard Manager safely removes the old replicas from downsized regions before Flux reclaims capacity.

The primary challenge with integrating stateful services today is that the default demand attribution algorithms do not always accurately capture requests costs. Such systems often exhibit interaction between requests, where processing of one request can affect the cost of subsequent requests. Our default attribution algorithms also do not capture persistent storage costs, the effects of caching, etc. We work with service teams to update our algorithms to better capture their capacity cost models. For example, TAO [16], Meta’s social graph store, maintains a custom cost model, which captures many of the above effects across their complex, distributed system. We integrate this cost model into Flux’s attribution models to correctly capture TAO’s capacity needs.

## 4 Design Alternatives

In this section, we discuss the major design alternatives.

### 4.1 RPC tracing

Flux relies on RPC tracing for gray box measurements of product-service capacity attribution. Meta has invested in a unified RPC stack [39], leading to high out-of-the-box tracing coverage without any additional instrumentation needed from service owners. Moreover, all our main traffic ingestion systems [30, 52] already implement sampled trace origination.

As of 2022, we have 52% of capacity usage covered by RPC tracing. For services that are not yet covered by RPC tracing, we have been working closely with the the service

owners to drive the adoption, because distributed tracing [40] as a fundamental capability in a large infrastructure has broad usage beyond capacity management, such as problem determination [17] and performance debugging [2].

Black box methods like statistical analysis [2] or heuristics [3, 38, 43] can be used to infer service call graphs without needing service-specific instrumentation. However, our highly interdependent microservice architecture makes employing such techniques less accurate. Since many of our backend systems are shared among multiple frontends, which invoke distinctly different callpaths, often with substantially different cost per request.

Black box methods were previously only evaluated on simple three-tier applications, while in our complex environment, the depth of call graphs reaches 14 and the RPC fanout is as high as 742, and hundreds of different upstream services may call a given service at varying call-graph depths. The full complexities of Meta’s service topology and call graphs are reported in detail in a recent work [29]. These complexities make statistical or heuristic methods less applicable to our environment.

Furthermore, because we can mandate high tracing coverage in our services, we can expect higher quality models, which in turn helps us provide greater levels of automation in global capacity management.

### 4.2 Nonlinear Service Models

The core service model used by Flux is linear: it assumes that capacity usage is linearly related to a chosen demand metric (see Equation 1) and a product traffic mix. While such models are simple to identify and to apply broadly, many services exhibit nonlinear capacity behaviors. When available, Flux can update its estimates by using more accurate nonlinear models such as those produced by load testing [55], queuing analysis [41], or by simulation.

Many of our services use continuous load testing to maintain an accurate model of the relationship between a service’s capacity usage and its RPC throughput. These models are used by our *Capacity Estimator* (CE) [14] to ensure that services are sized correctly for demand and remain disaster ready as determined by simulating various failure scenarios. However, simulations introduce nonlinearities that cannot be represented in a MIP assignment problem. To solve this problem, Flux invokes CE with the traffic distribution produced by the MIP solver. CE then runs its simulations against the proposed traffic distribution, and provides updated capacity estimates that incorporate planned failure scenarios.

We have found that using a combination of a default linear model for regionalization, along with a load-test-induced nonlinear model for improving estimates, works well in practice. Linear models provide upper bounds and are amenable to MIP optimization, while the nonlinear models provide higher accuracy, and usually operate within the bounds of the linear

models. Additionally, the simpler linear models are easier to explain and diagnose. Currently, 9.2% of services using Flux have load-test-induced models, and these services account for 46.4% of the total capacity allocated by Flux. This suggests that larger services are more concerned about capacity and are more likely to build their load-test-induced nonlinear models.

## 5 Discussion

In this section, we discuss the challenges of doing capacity planning in a complex real world and several ways of applying ideas in Flux to public clouds.

### 5.1 Practical Challenges

**Flawed baselines impact modeling accuracy.** Because Flux started with a baseline that was the result of many years worth of ad-hoc capacity management, the baseline itself does not reflect an ideal placement. Thus, when modeling capacity, we have to take extra care when interpreting model residuals: they could be due to imperfect modeling, or baseline itself not well-balanced. Flux provides *rebalancing* to service owners wishing to correct these imbalances in a controlled manner.

**Complete capacity models are hard to come by.** We have found that many capacity management practices rely on tribal knowledge, ad-hoc modeling, and implicit agreements between services. These are not captured in the service capacity models that Flux operates with, and thus cannot incorporate various de facto objectives or constraints. We work with service owners to codify these, but often find that we need to work around these with manual planning adjustments. We have also introduced tools like the capacity ledger (see §3.5) that help capture and mechanize previously ad-hoc capacity management practices.

These realities mean that there isn't a clear ground truth for capacity distribution, and require a nuanced interpretation of both modeling residuals and execution errors.

### 5.2 Applying Flux in Public Clouds

Many large public cloud customers maintain “virtual private clouds”, whereby they acquire large capacity pools of reserved instances. For example, Netflix has reported [36] that it runs thousands of services on hundreds of thousands of geo-distributed reserved instances [9] in AWS. These customers can apply Flux to manage these pools of reserved instances, and intelligently place services so that they are maximally utilized. These customers can also use Flux to extract recommendations about the type, location, and amount of capacity to acquire in order to accommodate growth, and to optimize the customer's capacity footprint.

Cloud providers could provide a new kind of capacity contract, whereby customers are guaranteed low-cost capacity, but are not guaranteed specific regional placement. The cloud provider offers an online capacity-planning tool through

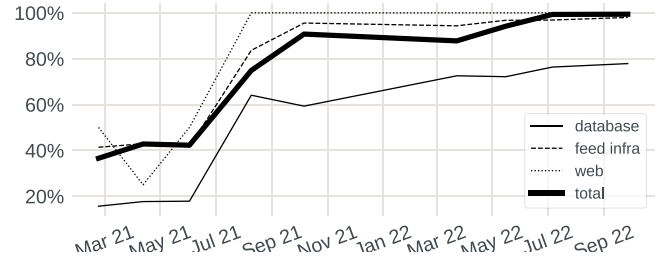


Figure 8: **Automated actions in total and by services.** Actions are nodes in the Flux's capacity placement plan, and include capacity and traffic management operation, such as adjusting regional quotas or resizing services.

which their customers continually update their aggregated placement constraints. The cloud provider then regularly re-regionalizes the capacity for customers that use this form of capacity, calling into customer's control planes to execute service and traffic rebalancing. This is similar to existing preemption APIs for spot instances [8].

## 6 Evaluation

Our evaluation answers the following questions:

1. How long does it take for Flux to execute its plan (6.1)?
2. Do Flux's service models help accurately assign global workloads to hardware in individual regions (6.2, 6.3)?
3. To what extent does Flux help meet the growing needs of out-of-region hardware refresh (6.4)?
4. How does Flux plan capacity and service placement for a specific service in practice (6.5)?

### 6.1 Execution Automation

Our goal for Flux is to maximize automation across both planning and execution, while incorporating humans-in-the-loop to review proposed actions and catch defects. We have granted Flux increasing autonomy as we gain confidence in the completeness and accuracy of Flux's models, its solvers, and automated execution systems. Currently, not all services support automation when adjusting their deployments across regions; Flux compensates by incorporating human-in-the-loop manual actions.

Figure 8 plots the degree of automation in Flux's execution plans, showing a handful of service groups as well as the overall automation. The drastic improvement in “total” around July 2021 was due to the introduction of a fully automated capacity distribution mechanism that integrates with our autoscaling system [14]. Over the past 2.5 years, we have increased automation in Flux from 27% to nearly 100%.

Figure 9 shows Flux's plan completion times. When Flux was first introduced, execution was dominated by operations requiring human feedback or execution. As we have simultaneously improved automation coverage and model quality,



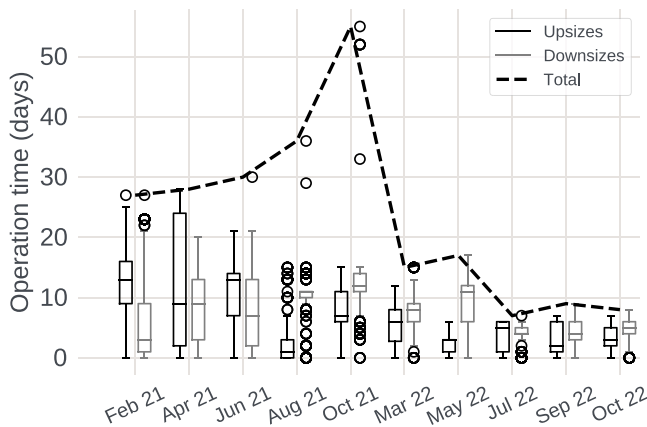


Figure 9: **Plan completion times.** The “total” curve represents the end-to-end plan execution time. The boxplots represent the distribution of individual service resize operation’s completion time. From bottom to top, the markers on a box represent, excluding outliers, the minimum, lower quartile, median, upper quartile, and maximum. Outliers are shown separately as small circles. The spike in the “total” curve represents a large shift in the complexity of Flux’s placement plans, which caused service automation coverage to lag Flux’s capacity coverage. We spent the ensuing months improving the coverage of our capacity automation tooling.

fewer operations require human-in-the-loop intervention and scrutiny, and most operations are now fully automated. Recently, executions take roughly 1 week. Even with model and automation improvements, some limits still remain. For example, cross-region data replication or cache warmup may still require long execution times for stateful services even if they support full automation.

## 6.2 Capacity Sizing Error

We define Flux’s capacity sizing *error* as the service capacity eventually used in production, minus Flux’s recommended capacity assignments, which include improvements from using load-test-induced models when available (§4.2). The errors exist for several reasons. First, since capacity-planning mistakes can be costly, service owners often review and sometimes revise Flux’s recommendation based on their domain knowledge of their services. Second, after Flux executes its capacity plan, our autoscaling system [14] may resize services in production, if it finds that additional capacity is needed to support Flux’s traffic shift, or that a service is left with a capacity surfeit.

Figure 10 shows the proportion of upsize and downsize capacity executed in each plan. A value of 100% means that execution was (in aggregate) exactly to plan. Over the course of the last year, we have improved planning accuracy significantly, primarily by working with service owners to improve their attribution and capacity models. We use execution his-

tory to incorporate expected error rates into Flux’s planning assumptions, and thus are able to tolerate this error by ensuring that we both (1) have sufficient capacity to support anticipated (aggregate) upsizes; and (2) are able to reclaim sufficient capacity where this is needed for refresh.

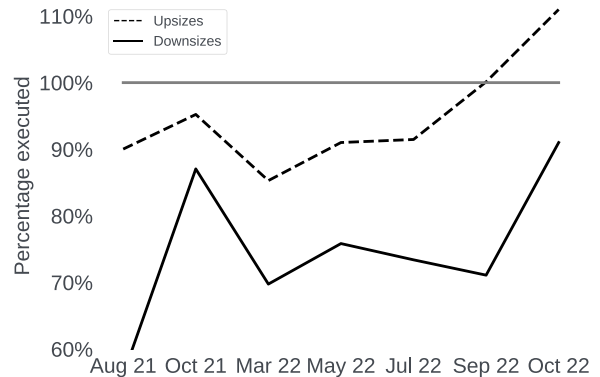


Figure 10: **Capacity-sizing error,** the percentage of Flux’s recommended assignments executed in production. Each data point represents the proportion of the total capacity in a Flux placement plan that was actually executed. The error is split by upsizes and downsizes.

Unless service owners explicitly opt out, we require them to review Flux’s placement plans through a UI tool. An Oct 2022 execution plan had 377 resize nodes. The total number of nodes available for Flux to execute on is about 3000. Of these, 81 nodes were for services that opted out of review; of the remaining 296 nodes, 84, or 22% of the total nodes, were revised by service owners. These revisions modify the resize node directly; Flux continues execution with the updated node. Our tool captures the reason given by service owners for each override, and we present the most frequent reasons below, with the number of each kind of override shown in parentheses.

**Insignificant capacity (15).** The service owner rejected the plan as the service does not have fully automated capacity management and the plan moved an insignificant amount of capacity. Therefore, the overhead to the service owner is too high to justify the benefits of execution.

**Service should not be rebalanced (13).** The service owner rejected the plan because the service should not be rebalanced by Flux. The remedy is to add the service to Flux’s execution blacklist.

**Insufficient headroom (10).** The plan would leave a service without enough headroom capacity, usually to accommodate anticipated growth. The remedy for this is to capture this requirement as a capacity event, so that it can be incorporated into Flux’s capacity plans.

**Deprecated service (8)** The service is deprecated, and should no longer be managed by Flux.

**Bad estimates(4)** The service owner judges the estimates to be incorrect, usually due to one of two reasons. First,

Flux has incorrectly attributed product demand to the service. In these cases, we repair the demand baselines, for example, by choosing another demand metric. Second, the linear model is inaccurate for the service. In these cases, we work with the service owner to adopt the load-test-induced model (see §4.2).

These overrides highlight the complexity of operating in a large-scale production environment. These results show that, with supervision, it is feasible for Flux to operate in a complex environment. Over time, as bugs are fixed and new features (e.g., headroom modeling) are added to Flux to cover a broader set of scenarios, we expect Flux to perform with higher accuracy and gain greater autonomy.

### 6.3 Model Residuals

The model *residual*  $\delta$  (expression 1 in §3.2.2) measures the portion of baseline service capacity distribution not explained by Flux’s service model. While the previously presented *error* metric reflects plan defects that could cause inefficient placement or operational risks,  $\delta$  merely reflects the imbalance between traffic and capacity distribution. Large residuals often reflect pre-existing capacity imbalance or inherent limitations in services which prevent the system from achieving an ideal balance.

Figure 11 shows the residual for several representative services. *Web product*’s residual varies between 3% and 5%. Such stateless services are usually well-modeled by Flux.

The residual for *feed infra* was initially higher at  $\approx 8\%$ , because its capacity distribution was uneven before Flux was applied. Over multiple placement cycles, we have used Flux to reduce this capacity imbalance, which is reflected in recent residuals that match those of *web product*. Due to limited capacity supply during COVID, Flux’s optimization objective has been dominated by supply constraints, and once Flux is able to meet decommission and growth objectives, we limit the infrastructure changes that Flux is allowed to introduce. As capacity supply improves in the future, we plan to use Flux

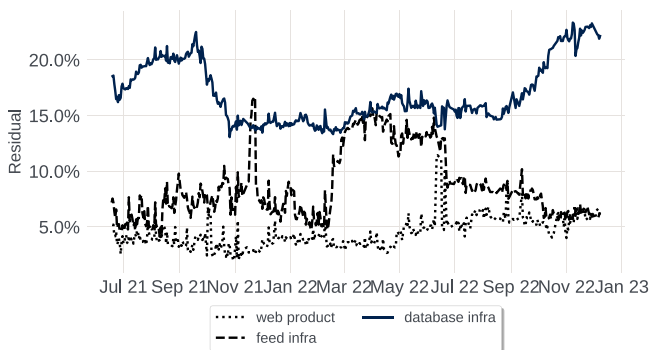


Figure 11: **Aggregate model residual.** We show  $\frac{\sum_{r \in R} |\delta_{s,r}|}{\sum_{r \in R} c_{s,r}}$ , for several representative services.

to more aggressively rebalance service capacity and reduce the residual. The short-term variance of the residual curves in Figure 11 correspond to load tests [6] and drain tests [52], as well as Flux traffic shifts. These events temporarily distort the relationship between traffic and capacity distribution, and are filtered out of Flux’s baseline.

Figure 12 shows the distribution of *feed infra*’s model residual across regions. This kind of plot guides us to work with service owners to improve their service balance by applying more aggressive balance objectives in Flux. The figure also demonstrates that, while the aggregate residual is higher at  $\approx 5\%$ , the per-region residual is generally less than 1%.

In Figure 11, *database infra*’s model residual is the highest due to some unique challenges associated with stateful services. For example, if a subset of hot data shards are the bottleneck, naively adding more capacity may not improve the service’s throughput proportionally. Many stateful services also have substantial capacity requirements for internal data replication [5], which fall outside of the usual request-response RPC regime, making it difficult to apply RPC tracing. We have been continuously improving Flux’s support for stateful services. In Figure 11, the initial reduction of residual from 22% to 15% was primarily due to improved attribution accuracy and coverage for *database*. The later regression coincides with deployment of new *database* systems into just a subset of regions, and for which we need to define new product attribution rules to capture correctly.

Overall, we deploy many stateful services, including databases, storage, and caches. Of the capacity currently managed by Flux, 14% is for stateful services. Our long-term strategy is to migrate stateful services to a common stateful framework called *Shard Manager* [32], which solves many common problems (e.g., hot shards) that impact stateful-service modeling. Moreover, *Shard Manager* is integrated with Flux so that the services it manages are automatically covered by Flux. Finally, *Shard Manager* intelligently places data shards onto the capacity allocated by Flux to minimize data-access latency [1,4].

### 6.4 Accelerating Out-of-Region Refresh

Figure 4 shows that as regions mature, they may have minimal power headroom to accommodate new hardware. Accordingly, the “quarterly OORR demand” data points in Figure 13 shows the rapidly increasing need for performing out-

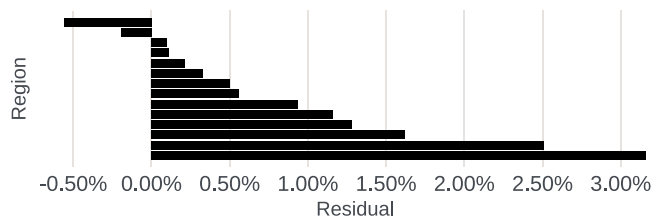


Figure 12: **Model residual for feed infra by region.**

of-region hardware refresh (OORR). Prior to 2020, a negligible amount of OORR was performed, and even then OORR was already a significant planning challenge. This was due to the lack of planning tools to compute global workload shuffling and traffic shifting for interdependent services, as well as lack of automation to execute such plans even when it was manually built. The benefits of Flux go far beyond OORR, but the imminent increase in OORR demand and the unsustainable toil in performing OORR motivated us to develop Flux. Flux has helped scale OORR planning and execution by  $\approx 950\%$  year-over-year. Currently, we perform global workload shifts once every 6 weeks; each shift typically reshuffling capacity for 100k-300k servers globally. The shifted capacity exceeds OORR demand because (1) the decommissioned capacity may not reflect the overall workload hardware composition, meaning that Flux must perform larger reshuffles to utilize the underlying hardware; and (2) Flux also allocates growth capacity and optimizes other infrastructure goals such as reducing disaster-readiness buffers.

### 6.5 Case Study: FeatureStore

As a detailed case study, we present the impact of Flux on FeatureStore, a flash-based key-value store serving features of machine learning models, deployed across thousands of servers, with a 99th percentile read latency of under 15ms, and a read request rate of 10s of millions per second. In September 2020, Flux was used for the first time to generate a service-placement plan for FeatureStore. Prior to that, its service placement was performed by humans.

A key event that took place during the September 2020 planning cycle was large-scale decommissioning of servers in three regions A, B, and C, resulting in a reduction of supply in those regions. Accordingly, we expect Flux to perform out-of-region hardware refresh and shift traffic away from those regions to others in order to accommodate this regional supply reduction, which is confirmed by Figure 14.

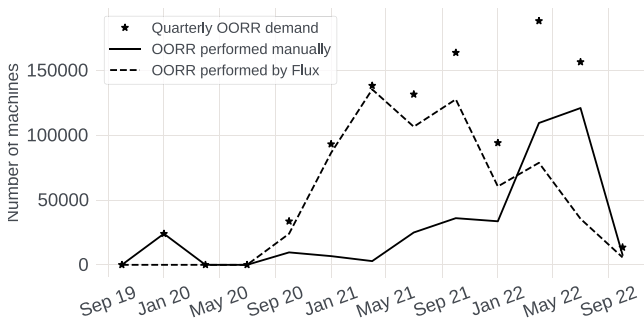


Figure 13: **Volume of out-of-region refresh (OORR).** This figure shows OORR planning and execution volumes handled by manual processes as well as Flux. The uptick in manual OORR in 2022 was due to a one-time large-scale decommissioning of data-warehouse hardware. Over all, Flux has helped scale our yearly OORR volume by  $\approx 950\%$ .

To understand Flux’s capacity assignment, we focus on two sets of services: (1) the frontend Web service that serves as the traffic gateway for FeatureStore, and (2) all backend services that support or consume FeatureStore. Figure 15 shows the ratio of capacity for these two sets, which varies due to different workload mixes across regions. Specifically, regions A and B show a lower capacity ratio, because they are data-warehouse heavy and have a larger FeatureStore footprint to support additional training workloads that are co-located with data warehouse but do not go through Web to access FeatureStore. Before Flux was applied to FeatureStore, capacity planners needed to explicitly take this into consideration, whereas Flux’s MIP formulation is able to automatically account for this and other factors affecting placement.

Recall from §3.3 that one optimization objective is to minimize deviation from the ratios in the globalized service model. This deviation is partially reflected in Figure 15 as the Web-to-FeatureStore capacity ratio’s variances across regions. Before Flux was applied to FeatureStore, the variances across regions C, D, and E were partially due to sub-optimal planning done by humans. Flux is able to reduce the variances by lowering the ratio for region D and increasing the ratio for region E, thus leading to better balance across regions C, D, and E. The improvement was small as this was the first time that Flux was applied to FeatureStore and was configured to be more conservative in introducing changes.

Deviation from ideal service capacity ratios is also reflected in the service’s unbalanced CPU utilization across regions. Note that, if FeatureStore’s capacity increase in a region is bigger than FeatureStore’s traffic increase in the region, we expect FeatureStore to have a lower CPU utilization

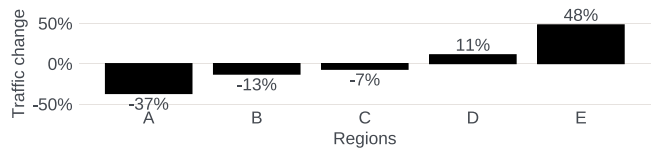


Figure 14: **FeatureStore traffic changes computed by Flux.** An example of how to read the figure: region A’s traffic change is  $\frac{\text{new traffic for FeatureStore in region A}}{\text{old traffic for FeatureStore in region A}} - 1 = -37\%$ . Only 5 out of 10s of regions are shown for readability.

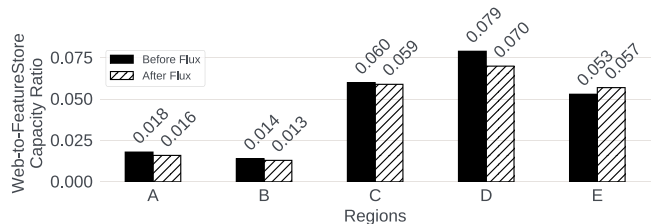


Figure 15: **Ratio of capacity for Web and FeatureStore.** Only 5 out of 10s of regions are shown for readability.

in the region after the change. This type of change can be applied to lower the CPU utilization of more heavily loaded regions, which is precisely what Flux did. For each region  $r$ , we calculate the average CPU utilization  $u_r$  of `FeatureStore` during its daily peak time window. Then we calculate the median of  $u_r$  across 10s of regions, and call it  $\hat{u}$ . Before and after Flux was applied to `FeatureStore`,  $\hat{u}$  was 55% and 50%, respectively. This indicates that Flux is effective in matching traffic distribution with capacity distribution to balance load across regions.

Figure 16 shows  $u_r$  for some sample regions. Overall, Flux reduces the CPU utilization of more heavily loaded regions such as regions A and D. In this instance, Flux was unable to increase CPU utilization in C due to the other constraints and objectives. This example shows that there are many factors to be considered during optimization, which is better suited to a MIP solver than humans.

## 7 Related Work

**Capacity management.** Capacity management impacts service performance and reliability as well as an organization’s capital and operating expenses. Two USENIX *login*: articles [27, 48] provide an overview of this topic. Misbah et al. provide a survey of resource management in federated cloud [33]. Several publications report capacity-management practices for internet services such as LinkedIn [53, 56], Uber [15], Google [34], and Netflix [36]. They focus on forecasting demand and capacity headroom, and are complementary to our work focusing on global service placement.

**Service tracing and modeling.** Several systems [13, 17, 30, 40, 46] insert unique request IDs into RPC calls to discover end-to-end service dependency. Our work adopts this approach. Some systems use statistical analysis [2] or heuristics [3, 38, 43] to infer service dependencies. Although these techniques are easier to deploy, they have not been proven to be robust enough to be used for internet-scale complex services. Both analytical models [51] and profiling techniques [41] have been applied to build performance models for three-tiered applications, but our environment is much more complex (see Figure 1). Endo *et al.* [20] call out the challenges of operating

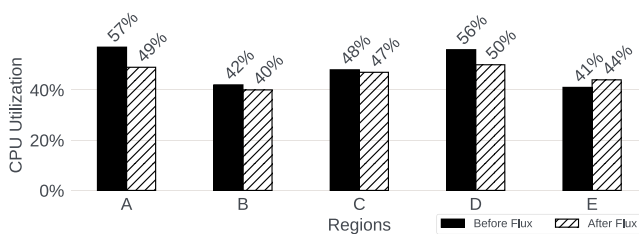


Figure 16: CPU utilization of `FeatureStore` per region before and after Flux is applied. Only 5 out of 10s of regions are shown for readability.

a distributed cloud, including resource modeling, but do not propose solutions.

**Service placement.** Yang *et al.* [54] propose joint service placement and traffic routing in mobile cloud, without considering service interdependencies. Malet and Pietzuch [35] propose placing services across datacenters to minimize network latency without considering the constraints of capacity supply and demand.

**Constrained optimization.** Constrained optimization has been used for resource allocation in different scenarios, including hardware-to-reservation assignment within a region [37], data-shard-to-container placement [32], and job scheduling within a cluster [19, 23–26, 42, 44, 49, 50]. None of them tackle the problem of global service placement.

**Infrastructure orchestration.** Cloud infrastructure orchestrators like Terraform [28] and CloudFormation [7] coordinate changes across multiple infrastructure systems in public clouds. These could be used to implement the orchestration component of Flux in a public cloud setting.

## 8 Conclusion

We identified the *regionalization problem* associated with managing customer services on large, global cloud footprints. We presented Flux, which solves regionalization by (1) using RPC tracing to build service regionalization models; (2) jointly solving service and traffic placement, growth capacity distribution, and infrastructure objectives; and (3) introducing a capacity orchestration system that safely and automatically rebalances services and traffic according to the computed plan. We shared our experience of using Flux at Meta’s large private cloud and discussed ways in which the ideas in Flux can be applied to public clouds.

## 9 Acknowledgements

This paper presents many years worth of work by multiple teams at Meta. They include: the Regional Fluidity, Capacity Engineering, Capacity Automation, Shard Manager, and Algorithmic Optimization teams. We would like to thank the current members of the Flux team who are not already on the author list: Kiryong Ha, Alex Cauthen, Chris Zheng, Hossein Tajik, Lin Xiao, Xiaomeng Shen, Austin Hendy, Daniel Boeve, Jikai Zhang, Tejash Shah, Kevin Lin, Partha Roy Chowdhury, Caroline Tony, Junjie Qian, Anand Saggi, Sebastiano Peluso, David Xu, Yichen Zhou, and Peter John Daoud.

We would also like to thank the following for their insightful comments, honest feedback, and partnership during the development of Flux: Maria Kacik, James Kneeland, Nasser Manesh, Haying Wang, Ariel Rao, Ash Shroff, and Alp Elci.

Finally, we thank Yunqi Zhang, Dimitrios Skarlatos, all reviewers, and our shepherd Z. Morley Mao for their help and insightful feedback.



## References

- [1] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.
- [2] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.
- [3] Animashree Anandkumar, Chatschik Bisdikian, and Dakshi Agrawal. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 133–144, 2008.
- [4] Masoud Saeida Ardekani and Douglas B Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–381, 2014.
- [5] Anonymized Authors. Meta’s Geo-Replicated Pub-Sub Service. 2015. Research paper published at a conference.
- [6] Anonymized Authors. Live traffic load testing in production at Meta. 2016. Research paper published at a conference.
- [7] AWS. Provision Infrastructure As Code, 2021. <https://aws.amazon.com/cloudformation/>.
- [8] AWS EC2. Use Capacity Rebalancing to handle Amazon EC2 Spot interruptions. <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-capacity-rebalancing.html>, 2022.
- [9] AWS reserved instance, 2021. <https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-reservation-models/amazon-ec2-reserved-instances.html>.
- [10] AWS user. Instance does not start—AWS out of capacity, 2016. <https://answers.sap.com/questions/12184202/instance-does-not-start---aws-out-of-capacity.html>.
- [11] AWS user. Capacity shortage hits AWS UK micro instances, 2017. [https://www.theregister.com/2017/03/24/aws\\_uk\\_t2\\_micro\\_instances\\_run\\_out/](https://www.theregister.com/2017/03/24/aws_uk_t2_micro_instances_run_out/).
- [12] AWS user. Hit with “insufficient capacity” for 3 days, 2018. [https://www.reddit.com/r/aws/comments/97rnvj/hit\\_with\\_insufficient\\_capacity\\_for\\_3\\_days\\_do\\_i/](https://www.reddit.com/r/aws/comments/97rnvj/hit_with_insufficient_capacity_for_3_days_do_i/).
- [13] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.
- [14] Daniel Boeve, Kiryong Ha, and Anca Agape. Throughput autoscaling: Dynamic sizing for Facebook.com, 2020. Blog post.
- [15] Rick Boone. “Capacity Prediction” instead of “Capacity Planning” How Uber Uses ML to Accurately Forecast Resource Utilization, 2020. SREcon20 Americas, <https://www.usenix.org/conference/srecon18americas/presentation/boone>.
- [16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook’s Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.
- [17] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.
- [18] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, Kevin Cole, and Dmitri Perelman. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.
- [19] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you’re late don’t blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC ’14, 2014.
- [20] Patricia Takako Endo, Andre Vitor de Almeida Palhares, Nadilma Nunes Pereira, Glauco Estacio Goncalves, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mangs. Resource allocation for distributed cloud: concepts and research challenges. *IEEE network*, 25(4):42–46, 2011.

- [21] Mary Jo Foley. Microsoft Azure customers reporting hitting virtual machine limits in U.S. East regions, 2019. <https://www.zdnet.com/article/microsoft-azure-customers-reporting-hitting-virtual-machine-limits-in-u-s-east-regions/>.
- [22] Mary Jo Foley. European users reporting they're hitting Azure capacity constraints, 2020. <https://www.zdnet.com/article/european-users-reporting-theyre-hitting-azure-capacity-constraints/>.
- [23] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.
- [24] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14*, 2014.
- [25] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.
- [26] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, November 2016.
- [27] David Hixson Kavita Guliani. Capacity Planning. *USENIX ;login.*, 40(1):49, 2015.
- [28] HashiCorp. Terraform by HashiCorp, 2021. <https://www.terraform.io/>.
- [29] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX, 2023.
- [30] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.
- [31] Kripa Krishnan. Weathering the unexpected: Failures happen, and resilience drills help organizations prepare for them. *Queue*, 10(9):30–37, sep 2012.
- [32] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.
- [33] Misbah Liaqat, Victor Chang, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Toseef, Umar Shoaib, and Rana Liaqat Ali. Federated cloud resource management: Review and discussion. *Journal of Network and Computer Applications*, 77:87–105, 2017.
- [34] Ramón Medrano Llamas. Capacity Planning at Scale, 2016. SREcon16 Europe, <https://www.usenix.org/conference/srecon16europe/program/medrano-llamas>.
- [35] Barnaby Malet and Peter Pietzuch. Resource allocation across multiple cloud data centres. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, pages 1–6, 2010.
- [36] Rajan Mittal and Andrew Park. Why Regional Reserved Instances Are a Game Changer for Netflix. In *AWS re:Invent*, 2017. <https://www.youtube.com/watch?v=i1EW6zmFbSM>.
- [37] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutorenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.
- [38] Patrick Reynolds, Janet L Wiener, Jeffrey C Mogul, Marcos K Aguilera, and Amin Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web*, pages 347–356, 2006.
- [39] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.
- [40] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver,

Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

- [41] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 71–84, 2005.
- [42] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.
- [43] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sri-ram Govindan, Bhuvan Urgaonkar, and Rong N Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *USENIX Annual technical conference*, 2009.
- [44] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, 2007.
- [45] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutorenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.
- [46] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):3–14, 2006.
- [47] Tim Anderson. ‘Azure appears to be full’: UK punters complain of capacity issues on Microsoft’s cloud, 2020. [https://www.theregister.com/2020/03/24/azure\\_seems\\_to\\_be\\_full/](https://www.theregister.com/2020/03/24/azure_seems_to_be_full/).
- [48] Luis Quesada Torres and Doug Colish. SRE Best Practices for Capacity Management. *USENIX ;login.*, 45(4):49, 2020.
- [49] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC ’12*, 2012.
- [50] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys ’16*, 2016.
- [51] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):291–302, 2005.
- [52] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.
- [53] Ruoying Wang, Lei Zhang, Yang Yang, Yi Zhen, Bo Long, Tie Wang, Vinoth Govindaraj, Todd Palino, Samir Tata, and Viji Nair. CapPredictor: A Capacity Headroom Prediction Framework in Cloud. In *Workshop on Cloud Intelligence, associated with Artificial Intelligence (AAAI 2020)*, 2020.
- [54] Lei Yang, Jiannong Cao, Guanqing Liang, and Xu Han. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers*, 65(5):1440–1452, 2015.
- [55] Wei Zheng, Ricardo Bianchini, G John Janakiraman, Jose Renato Santos, and Yoshio Turner. Justrunit: Experiment-based management of virtualized data centers. In *Proc. USENIX Annual technical conference*, pages 18–18, 2009.
- [56] Zhenyun Zhuang, Haricharan Ramachandra, Cuong Tran, Subbu Subramaniam, Chavdar Botev, Chaoyue Xiong, and Badri Sridharan. Capacity Planning and Headroom Analysis for Taming Database Replication Latency: Experiences with LinkedIn Internet Traffic. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 39–50, 2015.

$S$	Set of all services.
$R$	Set of all regions.
$H$	Set of all hardware types.
$P$	Set of all products.
$T$	Set of all timesteps.
$t_0$	Baseline timestep, i.e., infrastructure's current state.
$t_1$	Target timestep, i.e., the timestep for which we are executing.

### Inputs

$m_{r,h,t}$	Capacity pool: amount of type $h$ hardware available in region $r$ at time $t$ . It is generated by capacity forecast and includes the current capacity and net incoming and outgoing supply.
$\phi_{s,h,p}$	Globalized service baseline: output of the service-modeling process described in §3.2, indicating the fraction of service $s$ ' consumption of type $h$ hardware being attributed to product $p$ . We also scale the globalized service baseline by service growth, derived from our capacity budget management system, leading to varying values at different timesteps $t$ .
$\gamma_{s,h,t}$	The total growth capacity, indicating the global amount of type $h$ hardware allocated for service $s$ at time $t$ to support service growth. It is derived from Meta's budget management system, and is used to support product growth and launches.
$\tau_{p,t}$	The total traffic growth indicating the global increase in traffic to product $p$ at time $t$ , represented as the percentage above 100% global baseline traffic.
$e_n$	The penalty coefficient for objective $n$ . Penalty coefficients dictate how tradeoffs are compared.

### Assignment variables

Variable	Baseline	Distribution	Description
$x_{p,r,t}$	$\xi_{p,r}$	$X_t$	Traffic assignment, indicating the fraction of traffic from Regionalization Entity $e$ assigned to region $r$ at time $t$ . $X_t := \{x_{p,r,t} : p \in P, r \in R\}$ Invariant: $\forall p \in P \sum_{r \in R} \xi_{p,r} = 1$
$c_{s,r,h,t}$	$\kappa_{s,r,h}$	$C_t$	Capacity assignment, indicating the amount of type $h$ hardware allocated to service $s$ in region $r$ at time $t$ . $C_t := \{c_{s,r,h,t} : s \in S, r \in R, h \in H\}$
$g_{s,r,h,t}$	-	$G_t$	Growth assignment, indicating the amount of additional type $h$ hardware allocated to service $s$ in region $r$ at time $t$ for the purpose of growth. $G_t := \{g_{s,r,h,t} : s \in S, r \in R, h \in H\}$
$d_{r,h,t}$	-	$D_t$	Deficit assignment, indicating the amount of additional type $h$ hardware needed in region $r$ at time $t$ . $D_t := \{d_{r,h,t} : r \in R, h \in H\}$
$s_{r,h,t}$	-	$S_t$	Spares, indicating the amount of unallocated hardware of type $h$ in region $r$ at timestep $t$ . $S_t := \{s_{r,h,t} : r \in R, h \in H\}$
$o_{r,h,t}$	-	$O_t$	Double occupancy capacity. See explanation for Expression 14. $O_t := \{o_{r,h,t} : r \in R, h \in H\}$
$r_{s,r,h,t}$	$\rho_{s,r,h}$	-	Model residual, the difference between the observed baseline capacity distribution and the capacity distribution implied by the globalized service baseline, $\phi$ , distributed according to the baseline traffic distribution $\xi$ . $\rho_{s,r,h} := \kappa_{s,r,h} - \sum_{p \in P} \xi_{p,r} \times \phi_{s,h,p,t_0}$

Table 1: Notation used in the MIP formulation. *Baseline* means the current state of the infrastructure.

## A MIP Formulation in Flux

This appendix presents the MIP formulation used by Flux.

The core of the formulation is an assignment problem, represented by the assignment variables enumerated in Table 1. Each variable shares a region ( $r$ ) and timestep ( $t$ ) dimension; while capacity related assignments also include dimensions for the service being assigned ( $s$ ) and hardware type of the assignment ( $h$ ).

Next, we present the MIP problem formulation and explain each expression. The MIP problem is to minimize:

$$e_1 \times \sum_{p \in P, r \in R, t \in T} |x_{p,r,t} - x_{e,r,t_0}| \quad (2)$$

$$+ e_2 \times \sum_{t \in T, p \in P} \max_{r \in R} x_{p,r,t} \quad (3)$$

$$+ e_3 \times \sum_{r \in R, h \in H, t \in T} d_{r,h,t} \quad (4)$$

$$+ e_4 \times \sum_{r \in R, h \in H, t \in T} o_{r,h,t} \quad (5)$$

$$+ e_5 \times \sum_{p \in P, r \in R, h \in H, t \in T} |r_{p,r,h,t}| \quad (6)$$

$$+ e_6 \times \sum_{r \in R} \left| s_{r,h,t} - \frac{s_{r,h,t}}{\sum_{r \in R} s_{r,h,t}} \right| \quad \forall h \in H, t \in T \quad (7)$$

Subject to:

$$c_{s,r,h,t_0} = \kappa_{s,r,h} \quad r_{s,r,h,t_0} = \rho_{s,r,h,t_0} \quad (8)$$

$$x_{p,r,t_0} = \xi_{e,r} \quad o_{r,h,t_0} = 0 \quad (8)$$

$$c_{s,r,h,t} \geq \sum_{p \in P} (x_{p,r,t} \times \phi_{s,h,p,t}) + r_{s,r,h,t} \quad (9)$$

$$r_{s,r,h,t} \geq \min\{\rho_{s,r,h}, 0\} \quad (10)$$

$$m_{r,h,t} + d_{r,h,t} = s_{r,h,t} + \sum_{s \in S} c_{s,r,h,t} + \sum_{s \in S} g_{s,r,h,t} \quad (11)$$

$$\forall p \in P, t \in T \quad \sum_{r \in R} x_{p,r,t} = 1 + \tau_{p,t} \quad (12)$$

$$g_{s,r,h,t} = \frac{\gamma_{s,h,t} * c_{s,r,h,t}}{\sum_{r \in R} c_{s,r,h,t}} \quad (13)$$

$$o_{r,h,t} = \sum_{s \in S} c_{s,r,h,t-1} - c_{s,r,h,t} [c_{s,r,h,t} < c_{s,r,h,t-1}] \quad (14)$$

$$s_{r,h,t} + \sum_{s \in S} g_{s,r,h,t} \geq o_{r,h,t} \quad (15)$$

Below, we explain the intuition behind the MIP expressions.

**Stability objective.** Expression 2 penalizes traffic shifts to reduce churn in the infrastructure.

**Disaster-readiness objective.** Services have a disaster-readiness buffer to cope with any single-region failure. Expression 3 minimizes this buffer, by minimizing the size of the largest region.

**Objective to minimize deficits.** Expression 4 minimizes the additional hardware needed. Technically, a feasible solution requires that  $\forall r \in R, h \in H, t \in T \quad d_{r,h,t} = 0$ , and we use a high penalty  $p_4$  to ensure that deficits are non-zero only if a solution requires it.

**Objective to minimize deviation from service model.** Pre-existing placement imbalance, attribution inaccuracies, or traffic routing imbalances can cause baseline capacity assignments to deviate from the service model. Expression 6 minimizes deviation from the global service model defined by the globalized service baseline.  $r_{p,r,h,t}$  is the residual of the service model, which is further discussed in §6.3.



**Objective to balance spare pool distribution** Expression 7 encourages balancing unused capacity across regions to reduce hardware stranding. Sufficient unused capacity placement can also act as a buffer for capacity estimation discrepancies.

**Baseline.** Expression 8 establishes timestep  $t_0$  as the baseline, i.e., the current state of the infrastructure.

**Product attribution ratio constraint.** Expression 9 ensures that each service is allocated according to the ratio imputed from the service model (see §3.2). The model residual  $r_{s,r,h,t}$  is used to offset the placement.

**Residual-regression constraint.** Expression 10 prevents model residuals from regressing. Specifically, negative residuals (i.e., underprovisioned services per the model) are prevented from becoming more negative, while positive residuals may not become negative. Together, objective 6 and this constraint cause Flux to better balance service utilization across regions. Flux provides fine-grained controls (per service, region, hardware type) that let the service owner tune how aggressively Flux is allowed to rebalance a service. §6.5 provides a case study of this benefit.

**Capacity-sufficiency constraint.** Expression 11 ensures that each region has sufficient capacity to support the capacity assignment, as determined by expression 9. This also assigns additional hardware as deficits, if required for feasibility. Unallocated capacity is assigned to the spare pool  $S_t$ .

**Full-placement constraint.** Expression 12 ensures that each RE is fully placed.

**Organic growth constraint.** When  $\tau_{p,t} \geq 0$  Expression 12 places additional traffic demand which Expression 9 then allocates the organic growth capacity to each service according to its globalized service model. Organic growth is used to model increased traffic where all dependent services must be sized up proportionally.

**Inorganic growth constraint.** Expression 13 distributes the growth capacity proportionally to a service's placed capacity. Inorganic growth is used to distribute growth capacity to an individual service such that dependent services don't necessarily need to be resized, such as product launches.

**Double-occupancy constraint.** We execute the capacity upsizes before capacity downsizes (see §3.4) and cannot count on future released capacity to fund an ongoing upsize operation. Expression 14 defines the amount of capacity needed during an upsize operation, whereas Expression 15 ensures a valid intermediate state by enforcing sufficient capacity is available to prevent an upsize from using still occupied capacity. Growth is given out as a final step of execution, and can be used during the upsize stage. Expression 5 minimizes double occupancy.

**MIP Solver Scalability.** Flux uses the MIP solver well within its scalability limit. Our latest service placement run generated a problem with 24K assignment variables and 36K constraints. Solving the problem took only 5 seconds.

# Defcon: Preventing Overload with Graceful Feature Degradation

Justin J. Meza    Thote Gowda    Ahmed Eid    Tomiwa Ijaware    Dmitry Chernyshev  
Yi Yu    Md Nazim Uddin    Rohan Das    Chad Nachiappan    Sari Tran    Shuyang Shi  
Tina Luo    David Ke Hong    Sankaralingam Panneerselvam    Hans Ragas  
Svetlin Manavski    Weidong Wang    Francois Richard

*Meta Platforms, Inc.*

## Abstract

Every day, billions of people depend on Internet services for communication, commerce, and entertainment. Yet planetary-scale data center infrastructures consisting of millions of servers experience unplanned capacity outages and unexpected demand for resources; how can such infrastructures remain reliable in the face of capacity and workload flux?

In this paper, we introduce Defcon, a system for improving the availability of large-scale, globally-distributed Internet services using graceful *feature* degradation. In response to overload conditions, Defcon enables site operators to gradually disable less-critical features in order to reduce resource demand. Defcon presents a common interface to product developers to define feature *knobs* that represent degradation capabilities. Defcon automatically tests knobs to understand each knob’s product- and infrastructure-level trade-offs. At Meta, we have used Defcon to improve global product availability in the face of worldwide demand-surges in addition to large-scale infrastructure failures.

## 1 Introduction

Large-scale, globally-distributed Internet services, such as those operated by Alibaba, Amazon, Google, Meta, Microsoft, and Netflix, power modern human life by providing access to communication, commerce, entertainment, and many other experiences. At the same time, rapid advances in finance, artificial intelligence, machine learning, and virtual/augmented reality have solidified the utility of Internet services for much of humanity for the foreseeable future.

Internet services consist of *features* – functional building blocks that make up a larger product. For example, a video product consists of a search feature, a playback feature, a recommendation feature, and so on. Features are hierarchical: A *top-level* playback feature may itself consist of a video quality feature and a closed-caption feature, for example. Features, and the *products* they make up, power Internet services.

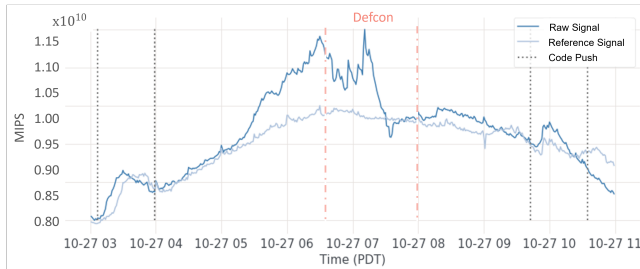
Products (and, by extension, features) run in data centers distributed around the planet. Analogous to the familiar von

Neumann architecture, computing at a planetary scale requires input/output (in the form of HTTP and RPC requests), computation (in the form of front-end servers), interconnect (the network backbone), caching, storage, and so on. Site operators deploy these resources within geographically-distributed data centers with the goal of ensuring that the workload *demanded* by users does not exceed the resources *supplied* by the network, servers, and so on.

Planning data center resources well requires predicting the future – or at least trying to. Capacity engineers rely on detailed user demand forecasts and server supply models to decide how and where to purchase and deploy resources, but alas, prophesy yet remains elusive: Errors and inaccuracy creep into models and forecasts, making data center capacity planning at times more of an art than a science. In addition, unpredictable world events – like global pandemics – can render even the most sophisticated predictions obsolete overnight.

At the end of the day, the people that use Internet services care about *availability*: Can they use the product that they want to use when they want to use it? Toward that end, companies work hard to ensure their products remain highly available. But what happens when things do not go according plan, such as during a persistent product demand increase due to a global pandemic, or when recovering from a global outage? Can we achieve high product availability without sacrificing additional resources? Can we be more efficient for rare – but inevitable – partial outages and survive them without additional server resources?

For example, Figure 1 shows a *real-world* surge in demand for one of Meta’s products, Facebook (measured on the y-axis in mega-instructions per second, or MIPS, executed by front-end servers for the product), that occurred over several hours on October 27, 2022. Localized peaks toward the left and right of the chart illustrate software deployment on the front-end systems, which consumes additional resources due to idle hosts updating their binaries and cold cache effects – these are expected behaviors. At around 5AM PDT, however, an unexpected increase in demand for the product happened to coincide with the daily peak usage of the product (shown



**Figure 1:** Defcon in action during a real-world site event. (See Section 1 for explanation.)

from a previous day as “reference signal”) leading to constructive interference and causing the product to run out of capacity and dangerously approach an *overload* condition (approximately  $1.15 \times 10^{10}$  MIPS) at which point fail-slow behavior and overload would occur. At around 6:35AM PDT (vertical dotted line), site operators engaged a system, which we present in this paper, in order to safely and efficiently reduce resource consumption (MIPS), *while still preserving access to core product functionality for all users*, avoiding an outage. Around 7:15AM PDT, as demand for the product continued to increase, site operators further engaged a next *level* of the system – leading to a correspondingly larger decrease in resource consumption, *bending the traffic (MIPS) surge curve* to restore it to nominal amounts of resource demand. After the surge had passed, at around 8:00AM PDT, site operators disengaged the system, restoring the product’s features to their original state.

In this paper, we present *Defcon*, a system to provide graceful *feature* degradation for Internet services. Defcon achieves high availability without sacrificing additional server resources by allowing site operators to *dynamically turn off product features* in response to rare (e.g., monthly or yearly) demand spikes or even unpredictable product demand increases. The key insight of Defcon is that *not all product features provide equal value* – many features can safely be turned off for short periods of time without altering a product’s fundamental behavior. Human guidance is used to define and actuate “knobs” – control flow annotations that represent the best capacity savings and user experience trade-offs.

We characterize the overload problem and solution space, apply a rigorous data-scientific methodology to analyze knob behavior, and describe a real-world at-scale testing methodology to validate the efficacy of Defcon. We also shed light on the design and organization of large-scale, real-world systems from the field as our approach accurately reflects the trade-offs involved in designing and implementing an initial solution to an emerging problem under realistic constraints. A key contribution of this paper is to prove the efficacy of feature degradation to help solve the overload problem in distributed systems.

We describe the design and implementation of Defcon and

our experience operating Defcon in production over the course of three years. We evaluate our approach using a combination of continual at-scale controlled tests as well as case studies from production incidents, including during a sustained demand surge. Overall, we find graceful feature degradation to be a powerful design pattern for system architects to efficiently improve the availability of large-scale distributed systems.

## 2 Background

Graceful degradation pervades the natural world: Removing ballast to prevent a ship from capsizing, escalators losing power and becoming ordinary stairs, starfish reproducing a lost limb, and so on. We observe analogous patterns of graceful degradation in the realm of computing and provide a brief overview of these techniques as well as a backdrop for why graceful degradation matters in large-scale Internet services, next.

### 2.1 Data Center Capacity Management

Modern hyper-scale data center infrastructures rely on server *capacity* distributed across the planet in order to support the diverse resource needs of the services that run in the data centers. Capacity Engineers rely on two inputs in order to make data center capacity planning decisions: *workload resource demand* and *server resource supply*.

Workload resource demand models the resource needs of a product in order to support its set of features. Capacity engineers normalize resources to a common unit baseline in order to plan resources across different server architectures or generations (e.g., Relative Resource Units, or RRUs) where resource types include computational throughput, storage capacity, memory bandwidth, and network bandwidth. Modeling workload resource demand involves understanding how many RRUs of different resource types are required to support product features. To accurately model future resource demand, engineers scale current resource demand based on feature growth projections.

Of course, in reality, resource supply and demand can behave in unpredictable ways. For example, a workload pattern change can change resource demand, while a data center outage can decrease resource supply. A key challenge, therefore, is *how to allocate resources in the face of constant infrastructure and workload flux*<sup>1</sup>. In many traditional systems, scenarios where resource demand > resource supply leads to fail-slow – and, eventually, overload-induced – system unavailability.

<sup>1</sup>Note that systems in Meta’s infrastructure are already equipped to automatically scale up and down capacity in response to predictable (e.g., diurnal) demand changes. Even so, there still comes a point when there is no remaining capacity to elastically expand a service into (such as during unpredictable load spikes or large outages).

Potential Solution	Description	Additional Resources	Engineering Effort	User Impact
0. Do Nothing	Allow overload to happen, leading to product outages.	None	None	Very High
1. Overprovision Resources	Increase server resources, leading to lower steady state utilization. Cannot fully predict future traffic patterns.	Prohibitively High	None	Potentially None
2. Drop User Requests	Reduce work by discarding user requests at a load balancer level (L4 or L7) before they enter into a data center.	None	Medium	High
3. Shed Server Load	Modify micro-services to decide when and which requests to drop for their service.	None	High	Medium
4. Degrade Product Features	Annotate control flow and avoid executing certain features on-demand.	None	High	Low

**Table 1:** Potential solutions to the overload problem and their associated trade-offs.

## 2.2 The Overload Problem

System *overload* occurs when the requested throughput of a service (e.g., measured in queries per second, QPS) exceeds the capabilities of the system, leading to a phenomena known as *congestion collapse* whereby *goodput* (a measure of the rate of successful responses from the service), decreases [2, 38]. Systems of any size can become overloaded, but the overload problem is especially acute in large, geographically-distributed Internet services, which can cause cascading failure scenarios, and can lead to widespread outages [12]. *Overload remains a fundamental problem in the operation of distributed systems.*

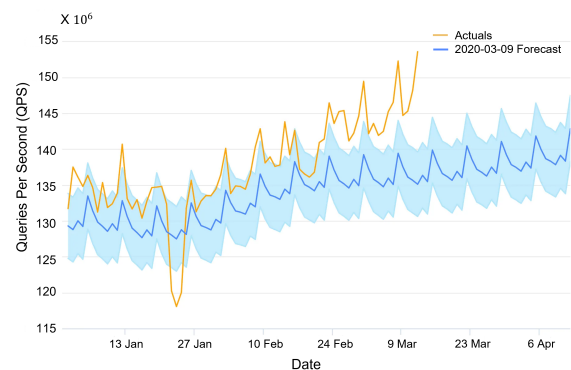
Meta’s infrastructure is organized around a collection of geographically-distributed data center failure domains, each representing around 5–12% of the overall capacity. Common failures such as bugs, network/power outages, and incorrect configuration happen within these failure domains and we have found 5–20% of savings to be a sweet spot for capacity savings for mitigating the risk of cascading failures due to overload. In this work, we assume a baseline of an overload-induced metastable failure state that leads to product outages for large portions of users for minutes to hours at a time.

Table 1 summarizes some potential solutions to the overload problem and how they trade off the amount of hardware resources (Server Resources), the amount of effort required of engineers to implement and maintain (Engineering Effort), and the potential impact to users (User Impact). For example, one way to attempt to solve the overload problem is to simply allocate more server resources for a distributed system (option 1). While potentially effective, simply allocating more resources can be inefficient, leading to low resource utilization when traffic is not at its infrequent (e.g., on the order of months or years) projected peak.

Furthermore, we can never perfectly predict traffic patterns and real-world events can often thwart even the best preparations. Take the COVID-19 pandemic as an example: In 2020, as more persons began to shelter in place, communication

that was once in-person began shifting to occur online. Figure 2 shows an example of how traffic for one product at Meta greatly exceeded its pre-pandemic resource plans. During global crises, Internet services often become more important than ever for humans to communicate and remain connected with each other. And while at Meta we were able to survive the COVID-19 demand surge, we wondered: “*Can we build systems that are inherently resilient in the face of unforeseen overload?*”

To answer this question, we found options 2–4 compelling. Note that options 2 and 3 both reduce work, but whereas option 2 reduces work at its *source*, option 3 reduces work at its *destination*. Specifically, for option 3, we considered a fine-grained backpressure-based approach, which led to noticeable user impact when capacity demand exceeded capacity supply and requests could not be processed. After evaluating the potential trade-offs at Meta, we opted for a technique to minimize user impact and found option 4, Degrade Features, to achieve the best trade-off: No additional server resources and low impact to users, albeit with an investment in engineering effort (which we qualify in Section 5).



**Figure 2:** Real-world events can often thwart even the most sophisticated preparation techniques, as shown by this graph of actual versus forecasted demand for Facebook in 2020 during the onset of the COVID-19 pandemic.



## 2.3 Related Work

Degradation is a self-adaptation technique that reduces the amount of work that servers need to perform to stay resilient during resource shortages, spiky load, and reduced hardware performance [12] that would otherwise cause a distributed system to enter a failure state [6, 16]. Degradation has been considered in many different contexts of system design, such as storage systems [8, 22, 31, 39], processor design [4, 17], cloud computing infrastructure [9, 11, 20, 25, 29], edge computing systems [21], web server applications [1, 2, 14], search engines [10], and mail services [27]. In network systems, graceful degradation is used to handle overload of network resources through intelligent connection management [36], traffic prioritization [3], traffic handover control [5], security hardening [15, 34], as examples.

Degrading the static content of a website was proposed in [2] and relevant techniques have been extended to dynamic content [26]. Degradation has also been proposed by cloud Infrastructure as a Service (IaaS) providers as a feature to increase cloud utilization [29] and used for adapting to the high network variability and possible network disruptions in edge computing infrastructure [21]. Defcon contributes to this area of research by developing a product-level, *feature-centric* framework to perform configurable *graceful* degradation of large-scale geo-distributed micro-services during spiky load and disaster events and providing real-world insights on how to build and operate such a system from its global of deployment in products at a large scale.

One alternative approach for surviving resource shortages during load spike or outage events is load shedding [11, 37, 38], which drops a proportion of load by dropping request traffic when a server approaches overload. However, load shedding sacrifices availability guarantees and broadly impacts user experience. In contrast, degradation techniques aim to provide high availability of products and services to users around the globe, which is critical to minimize impact.

Another area of related research is on specifying and realizing degradation for distributed systems. A relaxation lattice method was proposed for specifying the behavior of degradation [13]. Furthermore, specifications and implementations of degradation were presented in [39] as a complementary mechanism to fault tolerance in the design of highly-available distributed systems. Availability Knob [30] was proposed to provide a variety of availability guarantees, improving the utilization of reliability-heterogeneous infrastructures. In this work, we adopt the “knob” nomenclature, although for different means. Whereas Availability Knobs specify availability SLA flexibility, knobs as used in this work describe source code control flow annotations which can be enabled or disabled at-will.

Client-managed degradation was explored in the context of features like low power modes [18]. Our approach differs from client-managed degradation in three key ways. First, in

contrast to an ad-hoc approach to define individual points of degradation in client code (which, like a low power mode, then effectively become new “features” to maintain in the client), our approach provides a framework (knobs) that developers can use to efficiently encapsulate existing features, significantly reducing the development cost of degradation. Second, our approach provides developers with a framework to automatically test, analyze knob savings, and manage the lifecycle of knobs. Third, our approach extends to both client-side and server-side knobs, as it abstracts the knob control plane into configuration management as opposed to custom client (or server) code.

## 2.4 Graceful Feature Degradation

In this work, we ask the question, “Can we design distributed systems that remain available even when resource demand  $>$  resource supply?” While such systems would violate traditional system design assumptions, our key insight is that *not all features of a product are equally important* – if we can identify *essential* features (such as the ability to send a message in a messaging product) versus *fungible* features (such as an online status indicator for whether the message recipient is currently online), then we can gracefully *trade off fungible features for on-demand server resources*, while still preserving *essential features*. For example, the number of results can be considered as an adjustable *feature* for a search product.

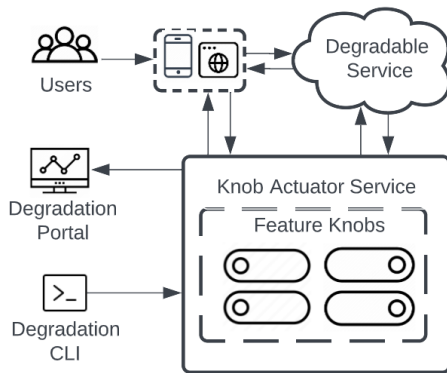
In this paper, we introduce the qualifier graceful *feature* degradation to refer to the property of a large, globally-distributed system to dynamically modify its behavior (features) in order to dynamically (i.e., *at runtime, without* recompilation or changing binary flags) alter its control flow for the purpose of reducing the system’s resource requirements. From here on, we use the terms “graceful degradation” and “graceful feature degradation” interchangeably.

## 3 Defcon

Defcon is a system to implement graceful degradation in large-scale distributed systems. Defcon is designed to be used during infrequent site emergency situations where demand is greater than supply. There are many reasons why a system may encounter situations where demand for the system’s resource exceeds the supply of resources for the system. Some examples are data center outages, load spikes during special events like New Year’s Eve, service overload due to a bug in a software deployment, and so on. We provide an overview of Defcon and discuss the design and implementation of its key system components, next.

### 3.1 Overview

Figure 3 shows an overview of Defcon. *Knobs* annotate program control flow eligible for degradation and follow a well-defined API. Product engineers use a Knob Definition Framework to annotate source code that can be degraded. These knobs are controlled using a Knob Actuator Service by site operators according to a policy. Not shown in the figure, a *Knob Testing Framework* registers knobs defined in the code base and automatically tests them to understand the sensitivity of product experience and resource consumption when the knob is turned on. We discuss each of these components next.



**Figure 3:** Overview of Defcon system architecture. Product software engineers use a *Knob Definition Framework* to label sections of control flow corresponding to specific features to conditionally execute. A *Knob Actuator Service* controls which knobs (and their corresponding features) are enabled or disabled in response to real-time events. Mobile clients, web clients, and micro-services all communicate via configuration with the Knob Actuator Service to dynamically determine control flow. A Degradation Portal provides insights for site operators to understand which knobs to enable in response to server resource shortages and a Degradation CLI allows humans to rapidly control knobs *en masse*.

### 3.2 Knob Definition Framework

A *knob* is a switch to enable or disable a feature in the code. Unlike feature flags, which require a binary restart in order to take effect, knobs are controlled dynamically while a binary is running. Knobs are implemented by a client library (or sidecar service) that determines the current state of each knob and are controlled using a configuration management system [32]. Software developers provide each knob a unique name, which they then can reference in their code. Thus, knobs can span multiple source code files, or even multiple binaries. Knobs come in two flavors:

**1. Server-side knobs** are implemented in binaries running on the servers in data centers. The advantage of server-side knobs is that we can adjust the knobs' state in seconds without any propagation delays.

**2. Client-side knobs** are implemented in client code running on phones, tablets, wearables, and so on. The advantage of client-side knobs is that they have the capability to reduce network load by stopping requests sent to the server along side reducing server load due to the request. Client-side knobs can also be controlled conditionally based on device metadata, such as cache state and network bandwidth availability. For example, Meta's mobile apps maintain a client-side cache response freshness threshold value. We update this freshness threshold value during Defcon to control incoming traffic. At Meta, we use server-side configuration to control these values. We use two approaches to propagate knobs state changes to clients, each with its own pros and cons:

**2.a. Silent Push Notification (SPN):** This approach uses a push notification system to propagate knobs state changes. At Meta, we have large numbers of client devices and the system takes around 30 minutes to finish all push notification jobs to propagate knobs state changes. SPN works like a typical app notification mechanism but instead of showing a notification to a user, the client app updates corresponding configuration fields.

**2.b. Mobile Configuration Pull (MCP):** In this approach, clients pull updated mobile configurations from servers through an API. At Meta, every client application implements two kinds of configuration-pull mechanisms: (1) A *full configuration pull* happens every 6 hours and pulls updated configuration data for every configuration definition. Full configuration pull is more thorough, but requires more network bandwidth and server resources. (2) During *Emergency Mobile Configuration (EMC) pull*, each client request triggers a server to inspect an emergency configuration file located on the server to fetch updated configuration data for the fields mentioned in the emergency configuration file. EMC consumes less network bandwidth and server resources, but requires manual intervention.

Listing 1 shows an example of defining a knob in Python (although APIs also exist for Rust, C++, Hack [35], and Java). Every knob has a unique name (with a namespace unique for each product name, `Feed` in this example), a *level* corresponding to the magnitude of resource reduction and used for grouping all knobs of a similar magnitude together, and a Boolean enabled state. The `export` statement instructs the build system to generate/update knob source code definitions in the code base.

**Listing 1:** Knob definition.

```
from configs.knobs import KnobConfig
disableCommentsRanking = KnobConfig(
    name = "Feed/DisableCommentsRanking",
    oncall = "owner_team_oncall",
    level = 2, # Impact magnitude.
    enabled = True)
export(disableCommentsRanking)
```

Listing 2 shows an example of using a knob in Python. To

use a knob, a developer must inspect the `enabled` field for the knob:<sup>2</sup> If the knob is disabled (the common case), the binary follows its normal control flow; if the knob is enabled (e.g., during an emergency), the binary follows a work-reducing control flow to reduce server resource consumption for every request served.

**Listing 2:** Knob usage.

```
from configs import ConfigReader
disableCommentsRanking = ConfigReader(
    "Feed/DisableCommentsRanking")
comments = fetchComments()
if (disableCommentsRanking.enabled == False)
    comments.RankUsingModel()
else # Knob enabled: do less work.
    comments.RankChronologically()
```

At Meta, knobs are not implemented haphazardly, but are instead carefully planned for by product teams with target resource savings set for different knob levels. Even so, the flexibility and ease of knob definition has enabled some products to implement and manage hundreds of knobs. Usually product teams choose the design of their knobs (i.e., server side knobs or client side knobs) based on the product behavior and the trade offs from controlling the demand at different places. Generally, knobs are defined at product feature level to stop the entire control flow across different surfaces.

Defcon knobs are added to both existing and new features. At Meta, features are deployed gradually with server-side controls and experiments. Meta’s deployment process requires product engineers to have a single-server side configuration to enable/disable their features. In Meta’s infrastructure, features are typically implemented as separate RPCs and therefore there is strong isolation between the control flow of each feature. For shared library code, product engineers have the choice to degrade at the library level or at a finer-grained RPC request level.

This process provides an advantage for developing knobs as a product team can simply extend these feature controls to check for Defcon configuration. Integrating knobs with feature development and deployment processes has other advantages: Ease of running experiments to test a knob for side-effects, measuring the capacity savings from disabling a knob, and measuring the impact of a knob on users (Section 3.4). User impact is then used to classify a knob into the correct Defcon level (Section 3.5). Once a product engineer is satisfied with a knob’s behavior, they will explicitly choose to include it in the Defcon system.

To aid product teams in understanding the breadth and behavior of the knobs they have defined, a browser-based graphical user interface is provided to help developers understand target level resource saving expectations, manage knob

<sup>2</sup>Knob configuration state is cached within memory on the server a binary is running on and accessed either by a shared library or a sidecar binary, typically requiring no more than microseconds to access.

metadata, visualize knob savings against the target expectations, and understand any user experience trade-offs (using a measurement methodology we discuss later). In turn, emergency responders use this user interface to understand Defcon level savings and the associated impact of enabling knobs.

### 3.3 Knob Actuator Service

We believe it is important to have a highly reliable tool with minimal dependencies to control Defcon, so that we can use Defcon even when most other systems are unavailable. The Knob Actuator Service is responsible for enabling or disabling (actuating) sets of knobs. Knobs are grouped into three categories: (1) By service name, (2) by product name, and (3) by feature name (such as “search,” “video,” “feed,” and so on).

The Knob Actuator Service also manages metadata for knobs, stored in a geographically-replicated relational (MySQL) database. Knob metadata includes: (1) The engineering oncall responsible for the knob’s definition, (2) the engineering team responsible for the knob’s usage, and (3) a cache of recent resource and user experience test results (discussed later in this section).

Finally, the Knob Actuator Service is responsible for changing the state of knobs. Knob state changes can be performed for individual knobs or for sets of knobs grouped using one of the three categories (service, product, or feature name). State changes occur in seconds for server-side knobs and in a couple of minutes for client-side knobs (due to the EMC pull cycle duration mentioned before).

While state changes across sets of knobs are used during site events that require additional capacity supply, state changes for *individual* knobs are used for testing knob impact. Knobs can be further selected for only a fraction of users participating in controlled A/B test experiments (discussed in the next sub-section).

At Meta, emergency responders receive notifications for various overload scenarios (including increased demand, decreased capacity, etc.) for services. The emergency responders use a Degradation Policy (defined in Section 3.5) to evaluate if Defcon can and should be used to reduce the load. Once the emergency responders decide on a course of action, they use capacity savings data from recent tests (which are available in a dashboard) to estimate what Defcon level should be enabled, and use the Knob Actuator Service to enable Defcon knobs to reduce the demand to the desired amount.

### 3.4 Knob Testing Framework

As an emergency response tool, we must test Defcon periodically to ensure its reliability and performance. Since Defcon will incrementally degrade product features when enabled, we go to great lengths to minimize its impact during testing.

Our strategy is to execute frequent, but *small scale* A/B tests to get continuous signals for Defcon knob resource savings as

well as potential issues, and infrequent large-scale exercises to validate these signals at scale and observe how the knobs for a product, service, or feature (and downstream services) behave when Defcon is enabled for all of the product's, service's, or feature's requests.

We classify production tests into two categories:

**1. Small scale tests.** A/B testing measuring user behavior metrics helps us quantify the impact on users and products. These tests are conducted across a product, but with a very small user base, (e.g., 0.01–2%) of a population over a certain duration (e.g., 15 minutes to 36 hours). The main goals from these tests are validating the knob set-up for the product, measuring the impact from the knobs, and measure the savings on downstream services using TRU. During A/B testing, we define four groups. One control group without any impact from Defcon and three treatment groups with different Defcon levels. We compare the resource consumption between the different groups with the control group as a baseline to measure the impact from Defcon on the service and the downstream services. To understand the impact of the Defcon knobs at a more granular level, we also run A/B tests at a service level (for any services that have knobs defined) and at a feature level.

We run server-based tests for multi-tenant backend services when per-user annotation is not propagated (e.g., for batch-processing services or multi-tenant services, where requests may belong to the system or several users simultaneously). In this case, we randomly select a small number of hosts for a particular service, and split these hosts in 4 groups, testing as described above.

We compare host metrics with the control group and store the results. The downside of this approach is that user experience may be momentarily inconsistent because consecutive requests from the same user may be served by different hosts. To minimise the user impact, we pick a negligible number of hosts for this test and run it for 5–15 minutes only. We run host-based tests weekly to always have fresh data and make sure that results are consistent. If results are not consistent, we adjust the number of randomly selected hosts. To make sure that our results are statistically significant and reliable we check that they match with empirical *large scale test* results (discussed next).

**2. Large scale tests.** Since Defcon is an emergency tool, we must test Defcon at scale to ensure its reliability and performance. We execute a Defcon service test on 100% of users quarterly to measure the resource savings at the product and service level and the demand reduction on the downstream services. Since during an emergency situation we may need support from Defcon for multiple products at the same time, we also execute combined degradation tests for multiple products together to measure the impact on Meta's infrastructure. During such tests, we enable Defcon knobs across products at levels 3, 2 and 1 for a short time and we monitor behavior similar to individual product tests.

### 3.5 Degradation Policy

Graceful feature degradation provides a trade-off between resource savings and product behavior. When designing the policy for when – and to what extent – to enable degradation, we must understand the trade-off between capacity savings from enabling a knob or collection of knobs and the user or product impact that comes from doing so. Product teams are responsible for defining key performance indicators that are closely measured and monitored during tests. Infrastructure teams provide a distributed tracing framework [19] to measure resource savings not only on the product, service, or feature where the knob is implemented, but also along the transitive closure of services affected by the knob.

Meta implements a four-level Defcon policy scheme whereby smaller-numbered levels correspond to *higher* amounts of degradation. Levels can be applied across the same features supported by the Knob Actuator Service (product, service, and feature). To ground the policy in reality, care has been taken to design each level around handling a specific set of failure scenarios:

**Level 4 (L4)** is the default state: All knobs are disabled.

**Level 3 (L3)** is used for handling overload situations resulting from relatively small-scale load spikes such as those seen during New Year's Eve or sporting events like World Cup. The Level 3 target savings is 5% of a product's overall demand.<sup>3</sup>

**Level 2 (L2)** is used for handling overload situation that arise from full data center region failures. Target savings is 10% of a product's overall demand (but can vary up or down based on a product's data center deployment model).

**Level 1 (L1)** is used during rare emergency events such as unforeseen global system outages. Target savings is 20% of a product's overall demand.

For setting target level savings, recall from Section 2.2 that Meta's infrastructure is organized around a collection of geographically-distributed data center failure domains, each representing around 5–12% of the overall capacity, making 5–20% of savings to be a sweet spot for mitigating the risk of cascading failures due to overload.

The Knob Definition Framework allows product developers the freedom and flexibility to explore potential knob resource savings and trade-offs in order to arrive at a portfolio of knobs that attempt to maximize the resource savings while minimizing the potential impact to users. When setting these level targets, service owners will translate demand reduction numbers to whichever resource they bottleneck on, like CPU,

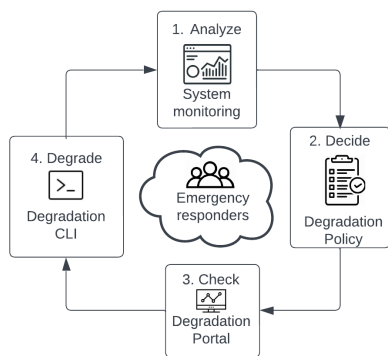
<sup>3</sup>Most front-end services at Meta have CPU utilization as the bounding resource, and so target CPU savings is the most salient metric to focus on.



memory, network. We rely on historical data for the amount of demand increase typically seen during similar past scenarios.

Figure 4 illustrates the four-step policy that emergency responders follow when operating Defcon in production. Emergency responders:

1. *Analyze* the state of the product resource demands and potential Defcon knob resource supplies using a system monitoring dashboard. The dashboard lists critical system resource utilization metrics described in Degradation Policy.
2. *Decide* if the situation can be mitigated by applying a Degradation Policy. A degradation policy specifies the resource and impact trade-offs associated with enabling knobs at a particular level for a product.
3. *Check* the current state of Defcon and adjust it in accordance with the desired Degradation Policy (e.g., enable L2 knobs for a product).
4. *Degrade* fungible product features using a command line interface (CLI). Continue at step 1, adjusting Defcon level as necessary.



**Figure 4:** Emergency responders rely on a well-defined *Degradation Policy* in order to engage Defcon effectively.

We next evaluate the efficacy and trade-offs associated with operating *Gratuit* in a real-world large-scale environment.

## 4 Evaluation

At Meta, we have operated Defcon across three products – Facebook, Instagram, and Messenger – for over three years. Defcon has been used to avert or avoid many dozens of situations that would have otherwise led to resource exhaustion and overload. We next evaluate Defcon to demonstrate its efficacy, both during tests as well as during real-world events.

### 4.1 Measurement Methodology

We relied on four main sources of data for our analysis:<sup>4</sup>

1. A *Real-time Monitoring System (RMS)* for measuring hardware counter statistics across the entire fleet of servers at Meta to measure real-time demand for server resources.
2. A *Resource Utilization Metric (RUM)* source of truth data set for available server resource supply, measured using load-test data. Supply metrics include available request throughput, CPU MIPS, memory bandwidth, and so on.
3. A *Transitive Resource Utilization (TRU)* system that uses a distributed tracing framework to measure resource changes across the transitive closure of services involved in serving requests from a particular service.
4. A *User Behavior Measurement (UBM)* framework for quantifying any user workload changes that occur during a test.

Using these systems, we measure two *system-level* metrics during testing: (1) The global savings in resource utilization on the product, service, or feature under test using RMS; and (2) the savings in resource utilization on back-end services that receive traffic from the product, feature, or service under test using a combination of RMS, RUM, and TRU.<sup>5</sup>

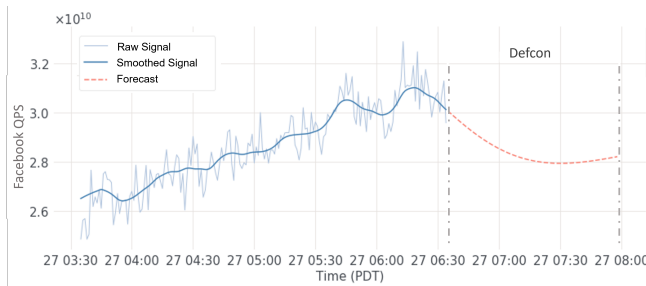
We rely on controlled UBM experiments in order to measure the non-system-level effects of Defcon in a statistically significant manner. Requests to a product, service, or feature under test are divided into two groups: A control group (group A) and a test group (group B). Resource usage and user behavior is measured and then compared between group A and group B. Tests are run on a small fraction of users (typically a fraction of a percent) and over a long enough period to obtain statistically-significant results (typically minutes to hours).

In addition, for large-scale tests that involve large collections of knobs, we utilize various data science approaches to model each of our metrics both before a test (a forecast) and after a test (a backcast). Through linear modeling and time-series forecasting/backcasting, we construct a source-of-truth signal during the test period. Resource savings are subsequently computed by taking the percentage difference between the real signal captured during the large-scale test and the constructed source-of-truth.

As an example of this methodology, Figure 5 shows the measured global request throughput for the Facebook product before a product-level Defcon test. This experiment was performed as the product was nearing its peak moment of

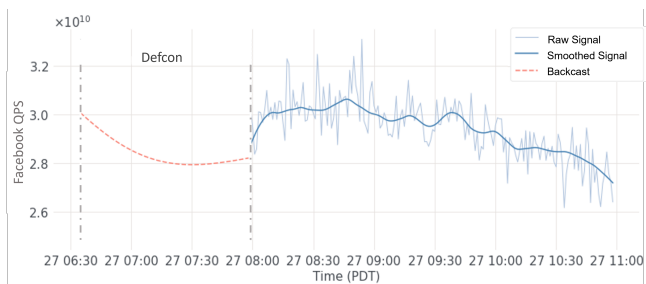
<sup>4</sup>Due to space constraints, we do not detail the design of these systems in this paper.

<sup>5</sup>Whether to use RMS or RUM depends on the resource consumption to measure.



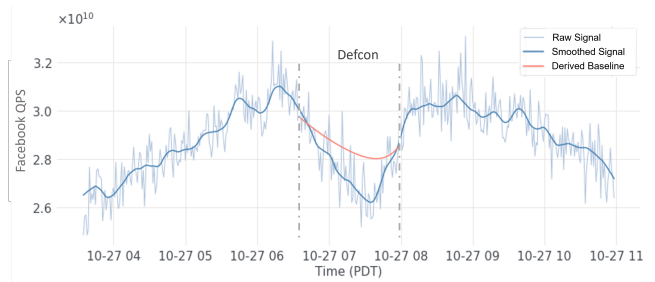
**Figure 5:** An example of timeseries forecasting. The measured global request throughput (QPS, y-axis) over time for the Facebook product immediately prior to enabling Defcon. A raw signal is converted to a smoothed signal and a forecast is generated from the smoothed signal.

request throughput (i.e., the highest organic load that we can test upon). Raw signal obtained prior to switching on Defcon (“raw signal”) is first smoothed (“smoothed signal”) and several time-series forecasting models are applied to obtain an estimate of what the raw signal would have looked like if Defcon was not turned on. Examples include linear, exponential, and Auto-Regressive Moving Average (ARMA) [28] models that are fitted using sections of the test signal before and after the Defcon degradation period.



**Figure 6:** An example of timeseries backcasting. Methodology is similar to Figure 5.

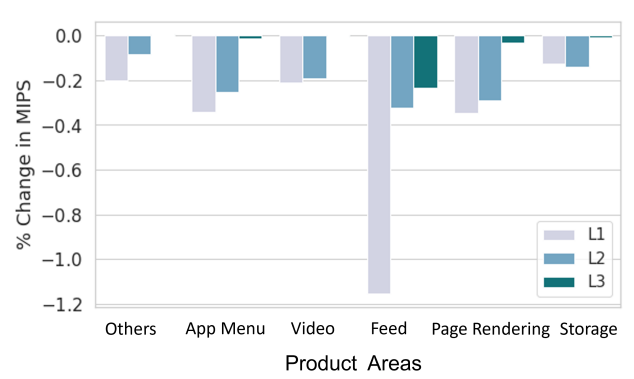
Aside from the forecasting models, we also reference past days’ signals in the steady state. The model which gives the smallest Median Percentage Error (MAPE) [23] is then chosen. Similarly, Figure 6 shows a time-series backcasting method applied to the smoothed signal gathered when Defcon is turned off. Note that the forecast and backcast use a somewhat conservative approach to ensure that measured savings are not over-estimated and to factor in headroom for the spikes observed in the raw signal. Finally, combining both the forecasted and backcasted signals (Figure 7), we derive a baseline which tells us what the metric would have been under normal circumstances when Defcon is not enabled. Savings are subsequently computed by taking the difference between the actual signal gathered during a Defcon test and the baseline.



**Figure 7:** By combining forecasts and backcasts during a Defcon test, we can construct a baseline to compare to the behavior when Defcon is enabled during a test.

## 4.2 Individual Product Tests

To continuously validate Defcon savings and reliability, we regularly perform A/B tests with a small percentage of users (0.01%, 0.05% and 0.5% for Level 1, Level 2, and Level 3 experiment groups respectively). The user percentages are set based on required population size of A/B test statistical analysis. Figure 8 shows the results of A/B test applied across different product areas. The y-axis shows the CPU resource consumption (measured in relative MIPS) and each bar corresponds to a group under test. As we expect, enabling lower levels of knobs generally results in more resource savings.



**Figure 8:** Results for Defcon tests for different sets of knobs (Product Areas) with different Defcon levels enabled. The y-axis plots the percentage change in CPU resource consumption, MIPS. Lower values indicate greater resource savings. Tests at each level are *inclusive* of higher levels (i.e., L1 tests also include L2 and L3 knobs). Notice that, generally, L1 knobs have larger resource savings than L2 knobs, and the same for L2 and L3. However, levels correspond to *impact* and not necessarily *savings*, and so some products, such as Storage, can achieve higher savings at lower levels of impact (L2 > L1).

Table 2 provides a detailed example of user impact metric data measured using the UBM framework described in Section 4.1 while testing at Level 1 for: (1) an individual knob, (2) a collection of knobs for a feature, and (3) all the knobs within all the features that make up a product. We observe

that degradation generally leads to relatively small user interaction changes, especially when compared to the alternative of an overload event leading to a site-wide outage. We also observe that enabling knobs can lead to user interaction *shifts* since user behavior changes in response to feature availability. For example, Video Watch Time at the Feature granularity increases when Level 1 knobs are turned on, as users engage in different ways to interact with a product.

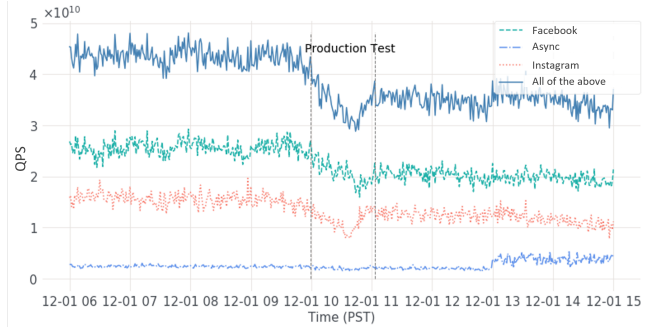
Metric Name	Knob	Feature	Product
User Interaction	-1.82%	-4.3%	-5%
News Feed Usage	-0.6%	-1.1%	-1.6%
Video Watch Time	-0.6%	+2.37%	-0.93%
App Usage Time	-0.36%	-1.9%	-11.0%

**Table 2:** Example UBM metrics when enabling Defcon Level 1 for a selected Knob, Feature, and Product. User Interaction measures high level user engagement metrics for an app, like the number of comments, reactions, posts, and so on over the test interval. News Feed Usage is a composite metric measuring feed views and feed interaction time. Video Watch Time is a composite metric aggregating time spent watching videos, count of live viewers, engagement with live videos, and so on. The Knob granularity is for an individual knob defined for the product. The Feature granularity is the feature that contains that individual knob and all other knobs that make up the feature. The Product granularity is for the product that contains that feature and all other features that make up the product.

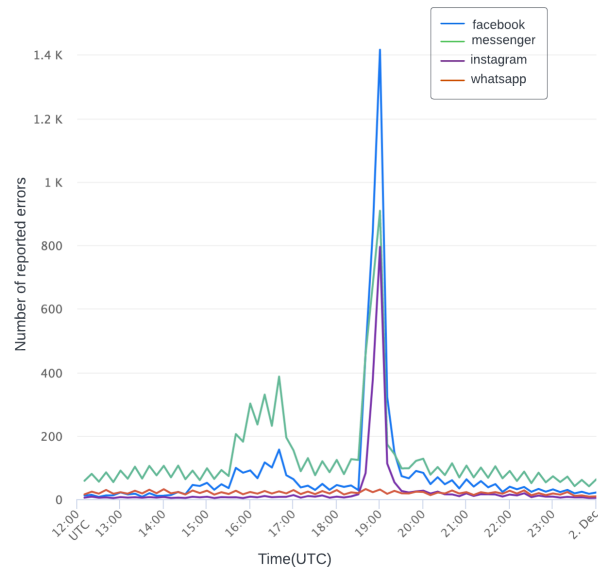
### 4.3 Combined Product Tests

At Meta, we regularly run combined degradation tests for multiple products. Figure 9 shows a combined Defcon test for three products: Facebook, a multi-tenant asynchronous compute platform (Async), and Instagram on 100% of traffic. The main goal of these tests is to accurately measure the combined transitive resource savings for shared backend services (here we illustrate the savings for Memcache, an in-memory key-value store [24]). As we can see, enabling Defcon across these three products leads to a compounding resource reduction for Memcache.

Of course, even when core product behavior remains unchanged, users may not expect to see changes in product features. At Meta, a user can submit a report when the user encounters something unexpected. Figure 10 shows user reports for four products during a combined test. As the figure shows, changing the features within products does not go unnoticed by users, with users submitting higher than nominal reports when Defcon is enabled. Note that this volume of reports – while keeping core product functionality available – is much preferred compared to fail-slow or overload conditions which could be several orders of magnitude larger without Defcon enabled.



**Figure 9:** An example of transitive resource changes on a multi-tenant backend system (Memcache), measured in QPS (y-axis). Requests are tagged according to which source of traffic sent the request: Facebook, an asynchronous compute platform (Async) and Instagram. We see that Facebook and Async contribute the most to the reduction in overall QPS (All of the Above).



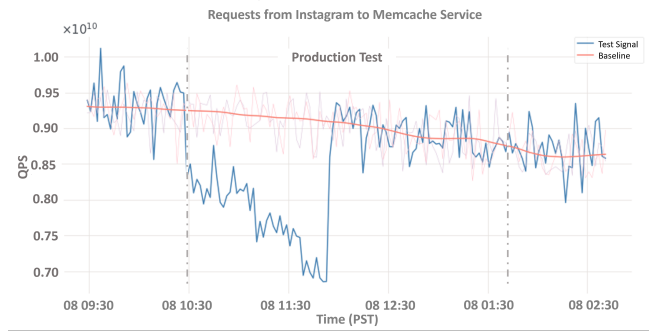
**Figure 10:** Number of reports submitted by users for different products during a combined product test.

### 4.4 Transitive Resource Savings

We next explore in more detail how transitive savings affect dependent services. Figure 11 shows an example of the resource savings achieved on the Memcache service (the same service from Figure 9) as measured *only* for the requests originating from the Feed product. Knobs of decreasing level were enabled incrementally during the test and then removed later in the test.

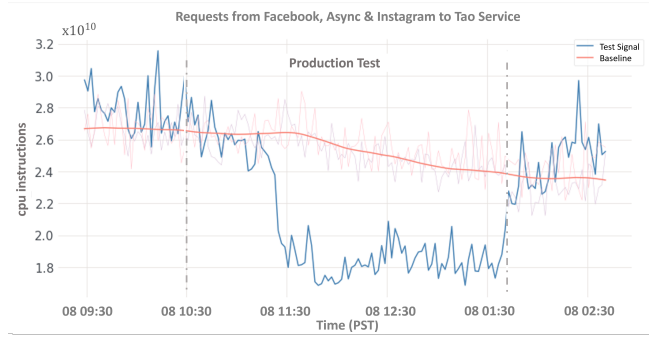
Note that knobs for the Feed product were only enabled during the first half of the annotated test range, and while other products participated in this test, using TRU, we were

able to observe the resource changes for only a single source of requests. Crucially, this savings is a beneficial *side-effect* of the reduction in workload from the front-end service and *not* a result of knobs defined in the Memcache service.



**Figure 11:** Resource savings, measured in QPS on the y-axis, on the Memcache service by enabling Defcon on upstream products, despite the Memcache service having no knobs defined. Reducing the request throughput to the Memcache service leads to corresponding reductions in resource consumption for the service *and its dependent services*.

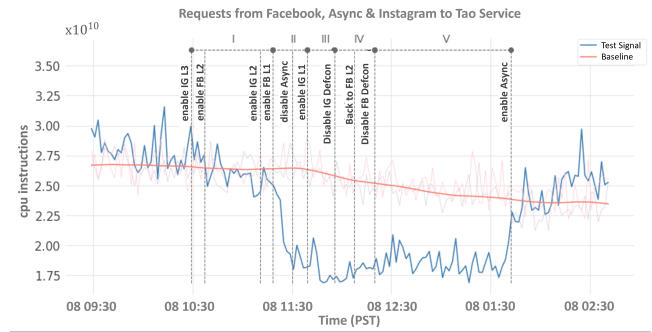
Figure 12 shows an example of resource savings for TAO (a social graph caching service [7]) when Defcon is enabled across the three products under test. In addition to showing another service that achieves savings despite not having *any* knobs defined, it also shows an example of how resource savings can remain relatively stable over long periods of time (hours).



**Figure 12:** Another example of transitive resource savings on a social graph (TAO) service that has no knobs defined. We find resource savings from Defcon to be stable over long periods of time (e.g., hours).

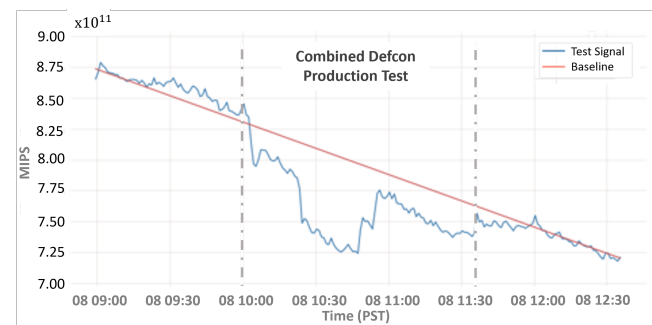
Figure 13 adds annotations to the results for TAO, showing the distinct phases involved in a large-scale test. As we can see, products enable knobs of decreasing level until reaching Level 1, remain at Level 1 for a small period of time, and then return to a disabled state. In this case, we can clearly see in Phase III, IV, and V that most of the demand for the TAO service comes from the Async product. We record insights

such as these as metadata for knobs and use the insights to inform decisions during real-world site events.



**Figure 13:** A detailed timeline of events during a typical multi-product Defcon test. This figure illustrates the complexity of testing Defcon at-scale in a production environment.

Interestingly, we also find that enabling Defcon across multiple products can achieve more resource savings for a product than enabling Defcon for that product alone. This occurs because some front-ends (such as the Facebook product) also serve RPC requests from other products (such as the Instagram product) so enabling Defcon on the other products reduces the resource consumption of the Facebook product. Figure 14 shows such an interaction for the Facebook product during a test when Defcon is applied to the asynchronous compute product, Async. We can see that even after Facebook knobs are disabled (around 17:50 UTC), Facebook still sees reduced resource consumption compared to its baseline due to reduced requests from the Async product.



**Figure 14:** An illustration of the inter-dependent resource savings of knobs: Enabling knobs for the asynchronous compute product, which sends requests to the Facebook product, leads to additive savings compared to enabling knobs for the Facebook product alone.

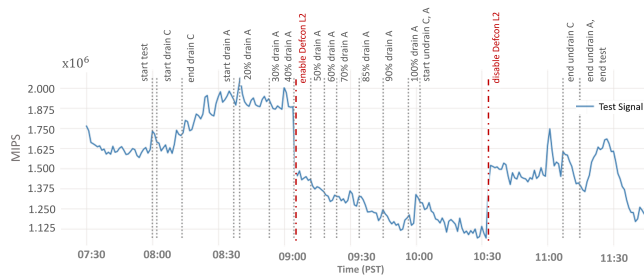
## 4.5 Outage Simulation Testing

At Meta we also simulate the conditions posed by large-scale outages such as natural disasters by redirecting traffic away from data center regions in order to concentrate more traffic



on the remaining regions, akin to what could happen during a fiber cut, hurricane, or power grid failure [33]. This reduces the available resource supply, effectively simulating load spike events such as New Year’s Eve or World Cup.

In Figure 15 we show the results of running a large-scale test on the Facebook product through the Facebook product’s peak moment of traffic. At the beginning of the test, we sequentially redirect traffic from multiple data center regions (labeled C and A) in order to concentrate enough load on the remaining regions. This operation continued until site operators began to detect a small volume (measured in very low parts per million of requests) of failed requests due to overload, whereupon Defcon was enabled at Level 2.

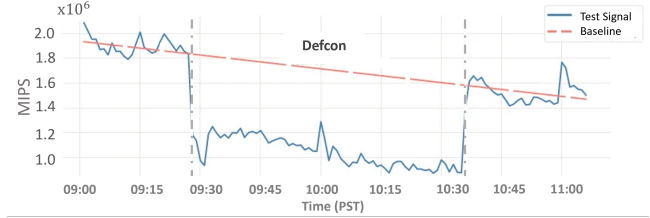


**Figure 15:** We regularly run tests to test the efficacy of Defcon under large-scale outages. In this test, we start by redirecting product traffic away from two data center regions in order to reduce the amount of server resource supply for the same amount of user demand, thereby increasing the resource utilization of the remainder of the fleet. We then enable Defcon in order to validate resource savings when products are in a highly-loaded state.

Moreover, after enabling Defcon at Level 2, we *continued to redirect traffic* until the second data center region was completely drained of traffic. This example illustrates how Defcon can effectively avert overload conditions that could ultimately lead to fail-slow behavior and wide-spread cascading failures. Tests such as this also provide valuable validation of the measured resource savings in a realistic environment: At-scale, at peak, and using the real production workload.

To illustrate the importance of at-scale testing, in Figure 16, we show an example of measured resource savings on a separate day, during a similar time, using the same knobs as the previous example. We can see that while the mean resource savings during this test is similar to the real-world increased load simulation, it is not exactly the same. A major reason for this is cold cache effects from traffic being redirected among data centers, a realistic concern during real-world outages.

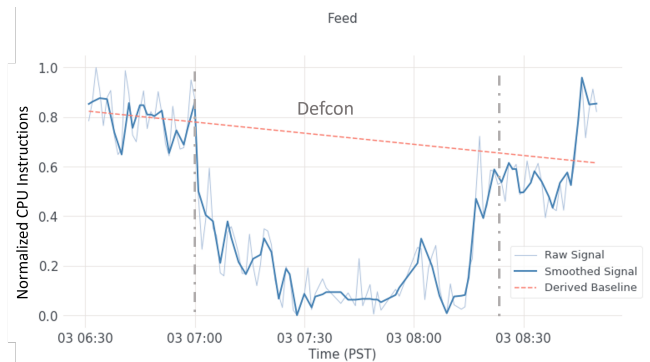
To illustrate the generality of our approach, Figures 17–20 show similar results across four different products – Feed, TAO, Memcache, and Graph Search – during a different two-data-center region drain test at peak levels of traffic with L2 knobs enabled. The y-axes of these figures have been normalized to compare the relative sensitivity to knobs across different products, with the measured savings corresponding



**Figure 16:** We find that resource savings (measured in MIPS on the y-axis) are *load-dependent*. In this example, having warm caches *increases* the amount of resource savings (corresponding to lower values of MIPS) compared to when outages are simulated (cf. Figure 15).

to 3.2% for Feed, 2% for TAO, 8% for Memcache, and 6% for Graph Search.

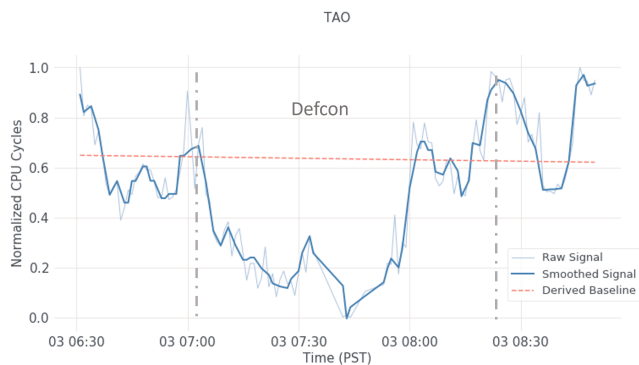
Notice that while different products achieve different levels of savings (these are reflections of their own target savings for L2 knobs), their response to enabling Defcon can vary due to caching effects and workload pattern changes in response to enabling knobs. The figures also illustrate how different products can customize their demand metrics used to measure and track their target Defcon savings (e.g., by using CPU Cycles or Power consumption).



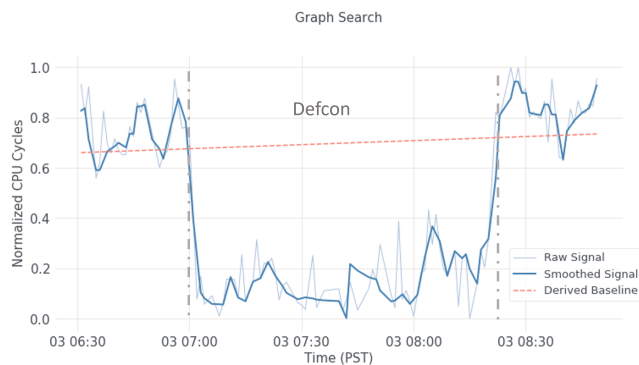
**Figure 17:** Results for L2 knobs enabled during a two–data-center region drain test for the Feed product.

## 4.6 Real-World Large-Scale Outage

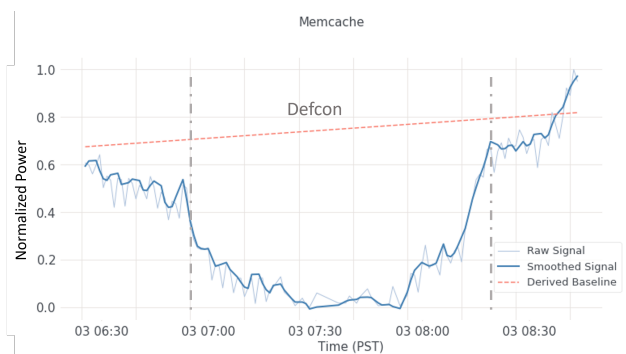
Since Defcon is an emergency tool used during large-scale outages, we must ensure that unknown unknowns are minimized. Based on the different degradation tests that we execute for products, and by measuring the impact on users and downstream services, we work closely with Site Reliability Engineers (SREs) to come up with degradation policies and guidelines for the scenarios where Defcon can help. During a real-world outage, SREs work with a lead emergency responder, the *Incident Manager (IM)*, who decides on which options from the Degradation Policy to pursue to mitigate an outage.



**Figure 18:** Results for L2 knobs enabled during a two–data-center region drain test for the TAO product.



**Figure 20:** Results for L2 knobs enabled during a two–data-center region drain test for the Graph Search product.



**Figure 19:** Results for L2 knobs enabled during a two–data-center region drain test for the Memcache product.

Figure 1 from Section 1, showed one such outage where the IM applied the principles listed in the Degradation Policy to avert a site-wide overload event and outage (please refer to Section 1 for an detailed explanation). During this event, the IM made the call to first engage L3 knobs for the product before eventually engaging L2 knobs. The fact that different levels of knobs – with different amounts of impact – existed, provided the IM with a spectrum of options to pursue in order to eventually arrive at the right degradation trade-off in real-time.

To ensure that we could mitigate this real-world event smoothly, we needed to ensure that the degradation policy discussed in Section 3.5 has been practiced by SREs and the IM. To make sure all the responding members are trained on using the policy, we frequently execute mock fire drills where we come up with potential scenarios, and role play the necessary steps to mitigate the risks. We have found such testing to be largely beneficial in ensuring emergency responders are prepared when disasters strike.

## 5 Lessons Learned

Over the past several years of using Defcon, we have learned several key lessons to consider for graceful feature degradation:

### 1. Understand business goals and customer perception to determine what to degrade.

Prior to implementing knobs, product engineers first decide on which features to degrade. Core product functionality must remain intact, but among the non-core features, we find that there exists a spectrum of resource savings compared to user impact. For this reason, product designers perform A/B tests (cf. Table 2) and make a decision about which knobs to keep and which to pass on. While this process requires human interaction, the Knob Definition Framework and Knob Testing Framework allow developers to quickly explore the knob definition space in order to determine the set of knobs that provide the most resource savings for the least user impact.

### 2. Leveraging graceful degradation during emergencies requires regular testing and an easy-to-consume understanding of the business and customer impact.

To provide an easy-to-consume understanding for emergency responders to use in the heat of the moment, product engineers provide a high-level *functional* summary of what is affected at each Defcon level. Using this summary, site incident managers can quickly determine whether enabling knobs for a product at a given level is an adequate response. Additionally, this summary benefits the public relations and communications team, who may need to respond to inquiries from customers or the media about product feature changes.

### 3. Degradation systems require high and regular commitment from product teams.

To motivate product engineers to work on Defcon knobs, we built mechanisms to provide recognition for investing in this technique for product reliability. We organize monthly Defcon meetings per product to showcase each team’s work to their organizational leader (e.g., a vice president). We also

leverage the concept of Defcon champions. A Defcon champion is someone who is passionate about reliability that can drive Defcon throughout the organization. Defcon champions identify and recruit people in their organization to work on Defcon.

#### **4. Knobs, once built, need to be regularly maintained.**

Implementing and maintaining knobs requires engineering effort. Identifying candidate features involves coordination with product developers to run experiments to understand the capacity savings and user impact of knobs. Developers, however, have provided feedback that controlling and testing knobs using a standardized framework has helped them to rapidly develop and deploy knobs. While automated systems measure and report knob behavior, regressions in capacity savings and user impact require manual investigation. We intend to explore automating this area of knob maintenance in future work.

#### **5. Low dependence and high availability actuation.**

To ensure that Defcon is ready to be deployed during disasters, we iterated on improving our operations and operational availability. As an example, we developed a CLI with minimal dependencies on other systems in our infrastructure to make sure that Defcon is ready to be enabled during partial failures and disasters. Having a low dependence and highly available mechanism for knob actuation is critical for facing real-world disasters.

#### **6. Developer experience and efficiency are key.**

Before Defcon existed, there were scattered independent efforts to try to achieve similar goals. By unifying these disparate efforts and providing tools to support teams in a structured manner, we were able to increase the coverage of Defcon and simplify knob maintenance. Since Defcon is built on top of existing tools at Meta, such as Configurator [32], developers do not need to learn new technologies to implement new Defcon knobs.

Safety is handled by ensuring that features are isolated at the RPC layer (a design practice at Meta) and thus knobs typically encapsulate control flow between RPC callers and callees. While fine-grained degradation within a binary serving an RPC request is possible, safety and consistency must be validated by product developers during initial knob testing. We note that such validation is similar to what developers must do when routinely modifying binary control flow (i.e., not for the purpose of Defcon knobs) – a common practice at Meta. To aid developers in knob definition, we provide guidance on how to properly implement and maintain knobs, as well as provide developers with a Knob Testing Framework to measure Defcon savings and track regressions.

The main challenge for developers in maintaining Defcon knobs is capacity savings regression tracking. Systems at Meta are constantly evolving, so the impact of existing knobs can drift over time. Because of this, we make sure that each

team tests Defcon savings at a limited scale in production at least once every three months (an interval chosen to balance knob impact with the need to understand behavior changes) using the Knob Testing Framework. We are actively exploring ways to test knobs more frequently at lower impact.

## **6 Conclusion**

We presented Defcon, a system for graceful feature degradation to prevent overload in large-scale Internet services. We hope that by characterizing the overload problem, the corresponding solution space, and our approach to graceful feature degradation, we will spark discussion within the research community about how best to tolerate overload-induced system behavior and advance reliable and available distributed system design.

## References

- [1] Trustworthy Graceful Degradation: Fault Tolerance across Service Boundaries. <https://www.usenix.org/conference/srecon21/presentation/rogers-prior>, 2021.
- [2] Tarek F. Abdelzaher and Nina Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31(11–16):1563–1577, may 1999.
- [3] Satyajeet Singh Ahuja et al. Network entitlement: contract-based network sharing with agility and SLO guarantees. In *SIGCOMM'22*, 2022.
- [4] S. Almukhaizim, T. Verdel, and Y. Makris. Cost-effective graceful degradation in speculative processor subsystems: the branch prediction case. In *Proceedings 21st International Conference on Computer Design*, pages 194–197, 2003.
- [5] Matteo Maria Aurizzi, Tommaso Rossi, Emanuele Raso, Ludovico Funari, and Ernestina Cianca. An SDN-Based Traffic Handover Control Procedure and SGD Management Logic for EHF Satellite Networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 196(C), sep 2021.
- [6] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable Failures in Distributed Systems. *HotOS '21*, page 221–227, 2021.
- [7] Nathan Bronson, Zachary Amsden, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. *USENIX ATC*, 2013.
- [8] Housseem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301, 2012.
- [9] Google Cloud. Infrastructure Design for Availability and Resilience. [https://services.google.com/fh/files/misc/infrastructure\\_design\\_for\\_availability\\_and\\_resilience\\_wp.pdf](https://services.google.com/fh/files/misc/infrastructure_design_for_availability_and_resilience_wp.pdf), 2020.
- [10] Shuai Ding, Sreenivas Gollapudi, Samuel Ieong, Krishnamurthy Kenthapadi, and Alexandros Ntoulas. Indexing Strategies for Graceful Degradation of Search Quality. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, page 575–584, 2011.
- [11] Google Site Reliability Engineering. Addressing Cascading Failures: Load Shedding and Graceful Degradation. [https://sre.google/sre-book/addressing-cascading-failures/#xref\\_cascading-failure\\_load-shed-graceful-degradation](https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degradation), 2019.
- [12] Haryadi S. Gunawi et al. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *FAST'18*, 2018.
- [13] M.P. Herlihy and J.M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.
- [14] Hideaki Hibino, Kenichi Kourai, and Shigeru. Difference of Degradation Schemes among Operating Systems — Experimental analysis for web application servers —. In *Proceedings of DSN 2005 Workshop on Dependable Software - Tools and Methods*, pages 172–179, 2005.
- [15] David Ke Hong, Qi Alfred Chen, and Z. Morley Mao. An Initial Investigation of Protocol Customization. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '17, 2017.
- [16] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.
- [17] Lin Huang, I-Hong Hou, Sachin Sapatnekar, and Jiang Hu. Graceful Degradation of Low-Criticality Tasks in Multiprocessor Dual-Criticality Systems. pages 159–169, 10 2018.
- [18] Xiaofan Jiang, Jay Taneja, Jorge Ortiz, Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, David Culler, Philip Levis, and Scott Shenker. An Architecture for Energy Management in Wireless Sensor Networks. *SIGBED Rev.*, 4(3), jul 2007.
- [19] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. *SOSP*, 2017.
- [20] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodríguez. Brownout: Building More Robust Cloud Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 700–711, 2014.



- [21] HyunJong Lee, Shadi Noghbi, Brian Noble, Matthew Furlong, and Landon P. Cox. BumbleBee: Application-Aware Adaptation for Edge-Cloud Orchestration. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 2022.
- [22] Jingqiang Lin, Bo Luo, Jiwu Jing, and Xiaokun Zhang. GRADE: Graceful Degradation in Byzantine Quorum Systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 171–180, 2012.
- [23] Spyros Makridakis. Accuracy measures: theoretical and practical concerns. *International Journal of Forecasting*, 1993.
- [24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry Li, Ryan McElroy, Michael Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkat Venkataramani. Scaling memcache at facebook. NSDI, 2013.
- [25] Alessandro Vittorio Papadopoulos, Jakub Krzywda, Erik Elmroth, and Martina Maggio. Power-Aware Cloud Brownout: Response Time and Power Consumption Control. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017.
- [26] Jeremy Philippe, Noel De Palma, Fabienne Boyer, and et Olivier Gruber. Self-adaptation of Service Level in Distributed Systems. *Software: Practice and Experience*, 40(3):259–283, 2010.
- [27] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, Availability, and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. *ACM Trans. Comput. Syst.*, 18(3):298, aug 2000.
- [28] Björn Schelter, M. Winterhalder, and J. Timmer. *Handbook of Time Series Analysis: Introduction and Overview*, chapter 1, pages 1–4. 2006.
- [29] Mohammad Shahradd, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. Incentivizing Self-Capping to Increase Cloud Utilization. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 52–65, 2017.
- [30] Mohammad Shahradd and David Wentzlaff. Availability Knob: Flexible User-Defined Availability in the Cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 42–56, 2016.
- [31] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Trans. Storage*, 1(2):133–170, may 2005.
- [32] Chunqiang Tang, Thawan Kooburat, Pradeep Venkat, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. SOSP, 2015.
- [33] Kaushik Veeraraghavan, Justin Meza, Sankaralingam Panneerselvam Scott Michelson, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. OSDI, 2018.
- [34] Jagannadh Vempati, Ram Dantu, Syed Badruddoja, and Mark Thompson. Adaptive and Predictive SDN Control During DDoS Attacks. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6, 2020.
- [35] Julien Verlaquet and Alok Menghrajani. Hack: a new programming language for hhvm. <https://engineering.fb.com/2014/03/20/developer-tools/hack-a-new-programming-language-for-hhvm/>, 2014.
- [36] J. Robert von Behren, Eric A. Brewer, Nikita Borisov, Michael Chen, Matt Welsh, Josh MacDonald, Jeremy Lau, Steve Gribble, and David Culler. Ninja: A Framework for Network Services. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, June 2002.
- [37] Eugene Wiehahn and John Walker. Target Group Load Shedding for Application Load Balancer. <https://aws.amazon.com/blogs/networking-and-content-delivery/target-group-load-shedding-for-application-load-balancer>, 2021.
- [38] David Yanacek. Using load shedding to avoid overload. <https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload>, 2020.
- [39] Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Roy Levin, and Chandramohan A. Thekkath. Graceful Degradation via Versions: Specifications and Implementations. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 264–273, 2007.

# Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback

Romil Bhardwaj<sup>\*1</sup>, Kirthevasan Kandasamy<sup>\*2</sup>, Asim Biswal<sup>1</sup>, Wenshuo Guo<sup>1</sup>, Benjamin Hindman<sup>1</sup>, Joseph Gonzalez<sup>1</sup>, Michael Jordan<sup>1</sup>, and Ion Stoica<sup>1</sup>

<sup>1</sup>UC Berkeley

<sup>2</sup>University of Wisconsin-Madison

## Abstract

Traditional systems for allocating finite cluster resources among competing jobs have either aimed at providing fairness, relied on users to specify their resource requirements, or have estimated these requirements via surrogate metrics (e.g. CPU utilization). These approaches do not account for a job’s real world performance (e.g. P95 latency). Existing performance-aware systems use offline profiled data and/or are designed for specific allocation objectives. In this work, we argue that resource allocation systems should directly account for real-world performance and the varied allocation objectives of users. In this pursuit, we build Cilantro.

At the core of Cilantro is an online learning mechanism which forms feedback loops with the jobs to estimate the resource to performance mappings and load shifts. This relieves users from the onerous task of job profiling and collects reliable real-time feedback. This is then used to achieve a variety of user-specified scheduling objectives. Cilantro handles the uncertainty in the learned models by adapting the underlying policy to work with confidence bounds. We demonstrate this in two settings. First, in a multi-tenant 1000 CPU cluster with 20 independent jobs, three of Cilantro’s policies outperform 9 other baselines on three different performance-aware scheduling objectives, improving user utilities by up to  $1.2 - 3.7\times$  and performs comparably to oracular policies. Second, in a microservices setting, where 160 CPUs must be distributed between 19 inter-dependent microservices, Cilantro outperforms 3 other baselines, reducing the end-to-end P99 latency to  $\times 0.57$  the next best baseline.

## 1 Introduction

The goal of cluster resource managers is to allocate a finite amount of scarce resources to competing jobs. When doing so, we should ensure that the allocations fulfill the users’ and the organization’s overall goals. Traditionally, resource allocation policies have aimed to provide fairness [16, 24], maximize resource utilization [61], maximize the amount of

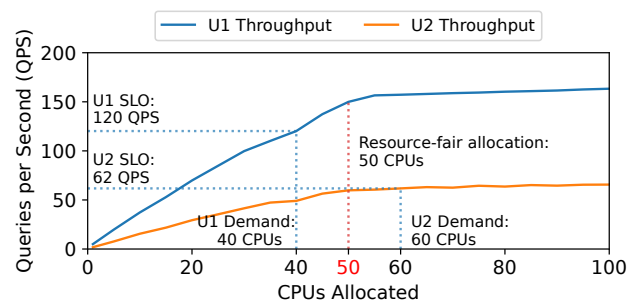


Figure 1: Two users, U1 and U2, serving TPC-DS benchmark queries with different resource-throughput mappings and performance goals (SLO). A user’s demand is the amount of CPUs needed for her SLO.

work done [24], or minimize queue lengths [47, 66]. However, these policies miss, or at best are imperfect proxies for what matters most to the users: the performance of their jobs in terms of real-world metrics that impact business (e.g. P99 latency or throughput for a serving job). Barring some recent exceptions [10, 18, 35, 64], resource allocation systems have traditionally focused on the resources requested by a job rather than the job’s real-world performance from using those resources (henceforth, simply *performance*).

To illustrate the pitfalls of performance-oblivious scheduling, consider an example where two users, U1 and U2, are sharing a cluster of 100 CPUs. They are each serving different sets of TPC-DS [43] queries and care about their throughput: U1’s service level objective (SLO) is 120 queries per second (QPS), while the U2’s SLO is 62 QPS. If the goal is to satisfy all user’s SLOs, how should CPUs be allocated? If it were known that the resource-to-throughput curves of the two users’ jobs were as shown in Figure 1, a scheduler can allocate 40 CPUs to the first job and 60 to the second. However, in practice, this mapping is usually not available and performance-oblivious scheduler will likely be suboptimal. For instance, a CPU-based fair allocation algorithm would allocate 50 CPUs to each user, which would result in U2 getting just 59 QPS, thus missing its SLO.

\* Co-primary authors.

Despite extensive theoretical work [16, 24, 28, 37, 38], performance-aware scheduling has remained challenging since the *resource-to-performance mappings* are usually unavailable in practice. To obtain these mappings, past work [15, 58, 64] profile their workloads before execution. Such profiling has three limitations. First, offline profiled resource-to-performance mappings may not reliably reflect a job’s performance in a production environment, as it may not capture the interference from other jobs [14] and the server’s performance variability [19]. Second, jobs’ resource requirements change with time due to varying load (e.g., arrival rate of external queries) and profiling typically cannot account for these changes. Third, such profiling is burdensome for users and expensive for organizations as it requires a large pool of resources to exhaustively profile a wide range of resource allocations. This informs the *first requirement* for this work: obtain the resource-to-performance mappings in the production environment where the job will be run.

Even if the resource-to-performance mappings are known, the choice of scheduling policy depends on the objective of the end-users (e.g. organization, developers). For instance, suppose in Figure 1, we wished to maximize the total throughput of the cluster, instead of trying to satisfy each user’s SLOs. In this case, we would allocate ~64 CPUs to U1 and ~36 to U2 for a total throughput of ~212 QPS. As more realistic examples, in multi-tenant clusters, we may wish to use policies which balance between performance and fairness [16, 24, 38]. In contrast, when we provision resources to different microservices of the same application, we are more interested in some end-to-end performance objective, such as application latency, and may wish to allocate more resources to critical microservices which bottleneck performance. These objectives can vary from organization to organization and optimizing for such different objectives requires different allocation policies. However, while end users may find it relatively easy to state their objective (e.g., satisfy all SLOs, maximize throughput), it is harder to design a policy to achieve it. This informs our *second requirement*: support a diverse set of user-defined scheduling objectives.

To address these requirements, we introduce Cilantro, a framework for performance-aware allocation of a single fungible resource type (e.g. CPUs, containers) among competing jobs (Figure 2). In Cilantro, end users first declare their desired scheduling objective. To satisfy the first requirement, a pool of performance learners and load forecasters analyzes live feedback from jobs and learns models to estimate resource-performance curves and load shifts for each job. To satisfy the second requirement, Cilantro’s scheduling policies, which are automatically derived based on the users’ objectives, leverage these estimated models to compute allocations for each job. As the learned models become accurate over time, Cilantro is able to eventually achieve the users’ objectives. This obviates the need for an offline model to estimate the required

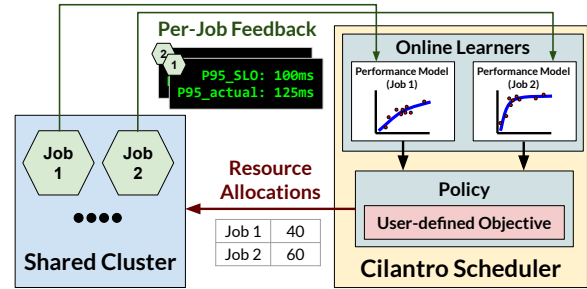


Figure 2: Cilantro overview. Cilantro uses continuous feedback to dynamically learn each job’s resource-to-performance mappings. An uncertainty-aware resource allocation policy, instantiated for the user’s objective, uses these mappings to determine allocations.

resource allocation for a given performance target, and allows Cilantro to optimize for custom objectives, such as various fairness or performance criteria. This is a marked departure from performance-oblivious policies, those based on unreliable proxy metrics such as CPU utilization and queue lengths, and other heuristic-based policies (using either surrogates [51] or performance metrics [10, 18]) which are designed for very specific scheduling objectives. Cilantro seamlessly enables the implementation of performance-aware policies in two settings: (i) multi-tenant resource allocation for independent jobs, and (ii) resource allocation for inter-dependent jobs (microservices) within an application.

Our proposed solution solves two key challenges. First, estimating resource-to-performance mappings online can be notoriously difficult due to highly stochastic nature of real-time production environments, unexpected load shifts, especially in the early stages when there is insufficient data. To operate without accurate estimates, Cilantro informs scheduling policies with confidence intervals of its estimates. Policies are designed to account for this uncertainty when making allocation decisions until the estimates become more accurate. Accounting for this uncertainty helps Cilantro conservatively explore the space of allocations making it robust to environment stochasticity and also to the idiosyncrasies specific to the performance models used.

Second, supporting a diversity of objectives in the same framework is challenging. The monolithic design of end-to-end feedback-driven approaches [34, 49, 64] restricts them only the objective they were originally designed for. Instead, Cilantro achieves generality in supporting custom objectives by decoupling the learning mechanisms from the allocation policy. This decoupling is necessary as it allows us to account for the effect of each job’s performance and load shifts on the objective individually. Moreover, this decoupling has other intangible benefits: it leads to a more transparent design which is easy to debug than monolithic systems which directly optimize for end-to-end performance, and if online job feedback cannot be obtained for a particular job, it is easy to swap the learners with profiled information or other sensible defaults.

We have implemented Cilantro as an open-source extension to the Kubernetes core scheduler, available at <https://github.com/romilbhardwaj/cilantro>. To evaluate Cilantro, first we deploy it on a 1000-CPU multi-tenant cluster which includes a diversity of real-world, latency and throughput-sensitive jobs. On three different allocation objectives, Cilantro’s policies are able to outperform 9 other baselines, and is able to compete with oracular policies which know the resource-to-performance mappings a priori on resource efficiency and fairness. When compared to resource-fair allocation, it is able to increase the performance of 1/3 of users in the clusters by  $1.2 - 3.7\times$ . Second, we evaluate Cilantro on a 160 CPU cluster where we wish to allocate CPUs to constituent microservices of an application. Here, Cilantro is able to minimize the end-to-end P99 latency of the application to  $\times 0.18$  the latency of a resource-fair scheduler and to  $\times 0.57$  of the next-best performance-aware baseline.

## 2 Background & Related Work

In this section, we compare Cilantro with prior work. Table 1 summarizes the key differences of Cilantro against other resource allocation systems and methods.

**Performance oblivious methods:** The simplest, yet popular approach to allocating finite resources among competing jobs, is to adopt a *resource fair* policy, which simply divides the resource equally (or proportional to weights) [2, 30, 32, 42]. As this does not account for jobs’ resource requirements, it is inadequate in all but the most trivial settings.

Several scheduling frameworks, such as Kubernetes [9], Mesos [29] and YARN [57], relies on users to submit their own resource demand. To execute resource allocations from policies, Kubernetes and YARN use resource reservations while Mesos negotiates through resource offers. This requires users to estimate their jobs’ resource needs, which can be difficult. They focus on one-way resource allocations and do not provide any mechanisms for the policy to get feedback on application performance. However, recognizing that end users may have varied scheduling objectives, these frameworks support and implement multiple policies.

**Methods based on proxy metrics:** The most common approach to account for resource requirements relies on proxy metrics (e.g. CPU utilization, work-queue lengths). Quasar [15] offline profiles jobs’ proxy metrics, and has a fixed operator-centric policy to maximize cluster utilization. Paragon [14] accounts for resource heterogeneity and inter-job interference to achieve performance guarantees. AGILE [46] models the resource pressure, and uses demand prediction to minimize SLO violations. The above works do not directly account for users’ performance goals and optimize for singular objectives.

**Methods which use offline profiling:** Some work has explored directly incorporating job performance via profiled his-

torical data. Morpheus [33] aims to mitigate performance unpredictability by defining SLOs and satisfying their resource demands by using models based on historical data. Ernest [58] provides methods for estimating performance curves using limited amount of data, but does not study using these estimates for resource allocation under scarcity. Sinan [64] partly uses profiled information for auto-scaling in a cloud environment. Quincy’s [30] min-cost flow formulation aims at providing fairness, but relies on offline estimates of data movement costs. For reasons explained in §1, offline profiling can be problematic and it is desirable to rely on real-time feedback to determine resource allocations.

**Methods which use online feedback:** Among related work, some feedback-driven systems account for performance metrics and SLOs in resource allocation. Jockey [18] focuses on meeting latency SLOs for a single job by modeling internal job dependencies to dynamically re-provision resources. Henge [35] defines new utility functions for stream processing workloads and aims to maximize a singular objective – the sum of utility of all jobs. [48] uses application hints in for prefetching disk blocks in the OS kernel. Gavel [44] is a scheduler for ML training workloads in heterogeneous environments with varying objectives. Since Gavel is focused on ML training, it’s policies are designed for throughput and a greedy optimizer computes the optimal allocation for each round. On the other hand, Cilantro supports any metric specified by the user and employs online learning to eventually converge on the optimal allocation. Finally, in a video streaming application, Minerva [45] studies methods for resource allocation so that all end users have the same quality of service. The highly customized policies used in the above works, while adequate to the allocation objectives set out by the authors, are not applicable for diverse cluster objectives which is our goal here.

**Variable resource amounts:** In other related work, PARTIES [10] allocates resources to jobs within the same server while always satisfying SLOs. If the SLOs of all jobs cannot be met, it evicts one of them to a different server; thus, it does not apply to our setting where there is a fixed amount of resources and eviction is not possible. Indeed, in §7 we show that a straightforward adaptation of PARTIES does not work as well. Sinan [64], DS2 [34], Autopilot [51] and FIRM [49] consider performance-aware resource allocation using online feedback when there is elasticity in resource availability, e.g. the cloud. Because these works can scale up to more resources than originally provisioned, they are not directly comparable to Cilantro which operates in a fixed cluster setting. While the cloud is an emerging use case, traditional fixed resource cluster management remains pertinent for privacy and cost reasons. Moreover, the above work focus on specific goals and are not designed to handle general allocation objectives. As an example, FIRM [49] focuses on autoscaling resources for single applications deployed as microservices to



	Cilantro	PARTIES [10]	Henge [35]	Autopilot [51]	Jockey [18]	Paragon [14]	Morpheus [33]	DS2 [34]	Quasar [15]	FIRM [49]	Sinan [64]	YARN [57]	Mesos [29]
Performance awareness	RW	RW	RW	RW	RW	RW	RW	RW	PM	PM	PM	PO	PO
Works without apriori performance model?	Y	Y	Y	Y	N	N	N	N	Y	Y	N	NA	NA
Supports multiple allocation objectives?	Y	N	N	N	N	N	N	N	N	N	N	Y <sup>1</sup>	Y <sup>1</sup>
Cluster size	Fix	Var	Fix	Var	Fix	Fix	Fix	Var	Fix	Var	Var	Fix	Fix

abbr. RW = Real-world metrics, e.g., latency, PM = Proxy metrics e.g., CPU util., PO = Performance oblivious, Fix = Fixed size, Var = Variable size  
<sup>1</sup> Supports multiple objectives, but only performance oblivious ones

Table 1: Cilantro and related work. Cilantro uses real-world metrics (e.g., latency) to build performance models online, which can be used to derive custom policies for different objectives.

minimize end-to-end SLO violations, Cilantro operates differently, reallocating a fixed number of resources according to user-specified objectives, which can include fairness considerations. Additionally, FIRM uses Reinforcement Learning with anomaly injection, in contrast to Cilantro, which focuses on resource-allocation under uncertainty and is agnostic to the learning method used.

### 3 Cilantro Architecture

Cilantro is a performance-aware scheduling framework that can optimize for various scheduling objectives without requiring any a priori knowledge of the resource-performance mapping of the workloads. The design of Cilantro is informed by the following two key insights.

**[I1] Offline profiling of resource-performance is insufficient.** Performance-aware policies rely on accurate estimates of resource-to-performance mappings and load shifts. Offline profiling of these resource-performance mappings can be inaccurate due to unpredictability in server and application performance [19] and changing traffic patterns [50]. Adapting to these changes necessitates continuously learning and predicting these unknowns in an online manner.

**[I2] Decoupling learning mechanisms and policies enables diverse scheduling objectives.** As different scheduling policies optimize different criteria, it may be challenging for a scheduling framework to generally support different policy types. Prior work on feedback-driven resource allocation [34, 49, 64] uses an end-to-end model for allocating resources for a fixed objective, such as total utility or cost. Optimizing for a different objective in these systems may require a complete redesign of the system and policy, or at the very least an expensive retraining of their models. Decoupling learning mechanisms from policies allows the model to be learned once and applied to multiple allocation objectives. This decoupling also increases transparency in the allocation decisions made by the scheduler and facilitates debugging.

We leverage these learnings to build Cilantro (Figure 3). Cilantro is composed of two key components: the central-

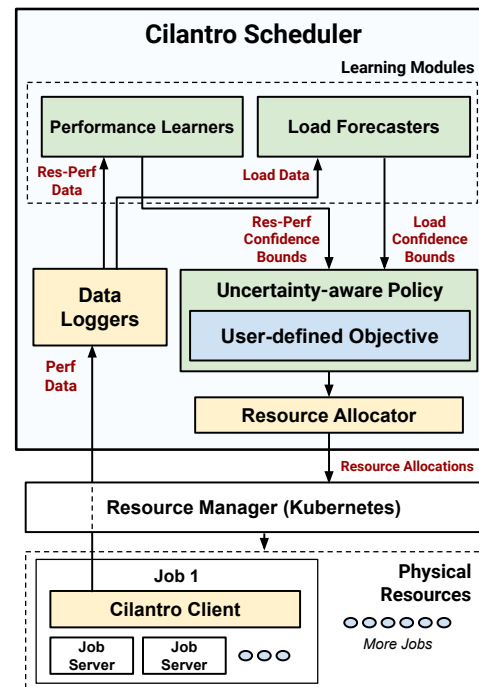


Figure 3: The Cilantro scheduler and client architecture. The scheduler generates resource allocations for jobs and the clients collect performance feedback to report to the scheduler.

ized Cilantro scheduler, which is responsible for generating resource allocations, and the Cilantro clients—lightweight sidecars co-located with each job—which fetch a job’s performance metrics and send them to the Cilantro scheduler. Informed by [I1], the Cilantro scheduler employs online learning to create increasingly accurate models of job performance and load. Guided by [I2], the policy optimizes a user-defined objective by polling these models for a resource-performance estimates to produce a resource allocation.

**Assumptions & terminology.** In this work, we will focus on jobs which can scale elastically with the number of resources with corresponding gains in performance. Examples of such workloads include stateless or stateful distributed

services (e.g., prediction serving [13], memcached [21], Cassandra [40]), distributed computation (ML training, MPI jobs) and distributed frameworks (e.g. Hadoop [52], Spark [62]). Some of these can be viewed as a collection of several tasks whose job size may vary with time, such as in serving jobs. Each task may refer to a query whose arrival rate may change with time. For jobs with such varying query rates, we will refer to the instantaneous rate of external query arrival as the *load* (measured in queries per second (QPS)). Finally, we assume there is a *fixed amount* of a *single, fungible* resource type that must be allocated.

**Cilantro scheduler:** The Cilantro scheduler is designed as a centralized asynchronous event driven system. Event sources include timers, performance updates received from the Cilantro clients, and cluster state updates from the underlying resource manager. Below, we describe the scheduler’s modules. Specific implementation details are available in §6.

**1. Data loggers.** Application metrics pushed from Cilantro clients are stored in memory-backed tables. They relay these metrics to the performance learners and load forecasters.

**2. Performance learner.** The performance learner learns a job’s performance as a function of the resource allocation and the load using an associated model. It periodically polls the data logger for new data and updates the model. The learner’s update frequency is constrained only by the velocity at which the model can be updated. One instance of a performance learner is maintained per application. A performance learner provides `get-perf-ucb` and `get-perf-lcb` interfaces for a policy, which return upper and lower confidence bounds for the performance as a function of the resources and load.

**3. Load forecasters.** In many real-world deployments, the job size could vary with time depending on the real-time traffic, which should be accounted for when allocating resources. The goal of the load forecaster is to estimate this load for the duration of a future allocation based on past observed loads via an associated time series model. It offers `get-load-ucb` interface for a policy which returns an upper confidence bound for the future load. Load forecasters are periodically updated by polling from the data loggers.

**4. Uncertainty-aware Policy.** Policies compute allocations in order to optimize for a user-specified scheduling objective. In an online setting, using direct estimates of the performance may fail as it does not reflect the uncertainty in the model. Therefore, Cilantro’s policies leverage confidence intervals of these estimates to account for this uncertainty in a principled manner when making allocation decisions (§4).

**5. Resource allocator.** The resource allocator is responsible for executing the resource allocations by interfacing with the underlying cluster manager. This module is driven via an allocation expiry event, upon which it invokes the policy’s `compute-alloc` method and allocates the resources. Allocation

expiry events are raised based on a timeout, resulting in a new round of allocations. In practice, the duration of an allocation round is limited by the agility of the environment. Since scaling jobs requires time, changing resource allocations too frequently can result in job thrashing (having to scale down before it has a chance to utilize new resources).

**Cilantro client:** The Cilantro client is a lightweight side-car container whose purpose is to poll the job to get its current performance, process it, and publish it to the scheduler’s data loggers. The primary task for the client is to extract metrics from their assigned job. Many systems expose REST endpoints to query system performance [3, 4], but often the applications also use monitoring tools such as Prometheus or Grafana. Depending on the job, the performance metric extraction logic is specified by the users. In §5, we describe built-in fallback options if job metrics are not available.

## 4 Policies

We now describe our policies for performance-aware resource allocation in two settings: multi-tenant resource allocation in a fixed cluster (§4.1), and allocating finite resources to constituent microservices of an application (§4.2).

**Set up & notation:** We will denote the number of jobs (or microservices) by  $n$ , the amount of resources by  $R$ , and an allocation by  $a = (a_1, \dots, a_n)$ , where  $a_j$  is the amount of resources allocated to job (or microservice)  $j$ . A scheduler should allocate these resources so that  $\sum_{j=1}^n a_j \leq R$ .

### 4.1 Resource allocation in shared clusters

Cilantro supports two classes of performance-aware allocation objectives in the multi-tenant setting: welfare-based, and demand-based. Our primary contributions are in §4.1.2 where we derive uncertainty-aware online variants of these policy classes. But first, we will review some common examples of such objectives in §4.1.1. For what follows, we will need to define the *performance*, *demand*, and *utility* of a job.

*Performance:* The *resource/load-to-performance mapping* (henceforth simply performance or performance mapping)  $p_j$  of a user’s job  $j$  refers to some raw metric of interest, which, say, can be obtained from a monitoring tool. We write the performance  $p_j(a_j, \ell_j)$  as a function of the resources received  $a_j$  and the load  $\ell_j$ . As we are allocating a single resource type,  $a_j$  is a single number, as is  $\ell_j$ . For example, in a serving job with a P95, 100 ms latency SLO, the performance may be the fraction of queries completed in under 100 ms, and the load may refer to the external arrival rate of queries.

*Demand:* If a job has a well-defined SLO, we define the *demand*  $d_j$  to be the minimum amount of resources needed to achieve this SLO. The demand depends on the job’s performance curve  $p_j$ , SLO, and load  $\ell_j$ .

*Utility:* The *utility*  $u_j$  of a job is the *practical value* derived

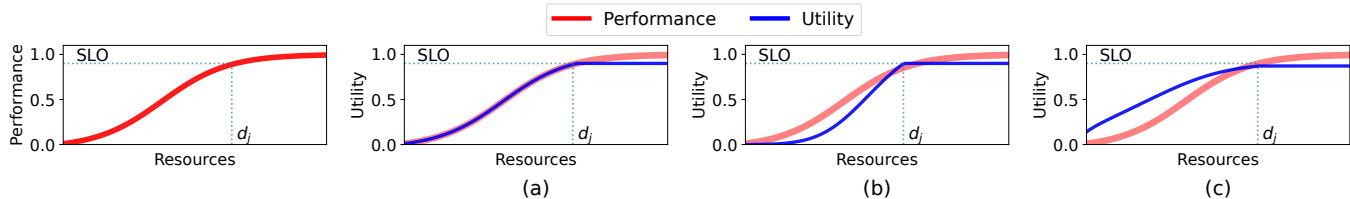


Figure 4: Three candidates for SLO-based utility functions. The left-most figure shows a job’s performance  $p_j$  as a function of the resources (for fixed load). In (a), the utility scales linearly with performance until the SLO, i.e.  $u'_j(p) \propto \min(p, \text{SLO})$ , whereas in (b) it scales quadratically  $u'_j(p) \propto \min(p, \text{SLO})^2$ , and in (c) it scales with the square-root  $u'_j(p) \propto \min(p, \text{SLO})^{1/2}$ . Here, (b) captures settings where even small SLO violations are critical while (c) captures settings where small SLO violations are not very significant.

due to its performance. Generally,  $u_j$  is a non-decreasing function of the performance and we can write  $u_j(a_j, \ell_j) = u'_j(p_j(a_j, \ell_j))$  for some non-decreasing function  $u'_j$ .

**Examples of utilities.** The simplest option is to set the utility to be equal to the performance  $u_j = p_j$ , i.e.,  $u'_j$  is the identity. However, we may also choose a utility which is more applicable when there are well-defined SLOs. Fig. 4 illustrates three candidates for  $u'_j$ : the maximum utility for any job is set to 1, which is achieved for any performance greater than the SLO; for performances below the SLO, we may set the utility to (a) decrease proportionally with SLO violation, (b) decrease sharply in settings where small SLO violations are critical (e.g., with external customers where SLO violations can lead to penalties [1] and a loss of credibility), (c) decrease gradually when small SLO violations are not critical (e.g., soft SLOs internal to an organization). Such utility forms which are ‘clipped’ at the SLO provide a simple way to compare jobs with heterogeneous performance metrics and SLOs, such as latency and throughput. Prior work have also used similar forms of utility [23, 35, 60]. For these reasons, our experiments also use these forms, although we emphasize that Cilantro can handle any utility form which increases with performance.

#### 4.1.1 Review of multi-tenant allocation when performance mappings are known

We will first review two classes of multi-tenant allocation objectives supported in Cilantro—welfare-based and demand-based—and three examples of such objectives. In §4.1.2, we will develop online learning policies that achieve the same objectives when performance mappings are unknown.

**Welfare-based objectives:** These policies aim to maximize a given cluster-wide *welfare* function  $W$ , which is a function of the utility of each job, i.e.,  $W = W(u_1, \dots, u_n)$ . Below, we describe two common welfare-based objectives.

(i) *Social welfare (a.k.a. Kelly mechanism [38]):* We choose the allocation  $a$  which maximizes the social welfare (the average utility), i.e.  $a = \text{argmax } W_S$ , where,

$$W_S = \frac{1}{n} \sum_{j=1}^n u_j(a_j, \ell_j) = \frac{1}{n} \sum_{j=1}^n u'_j(p_j(a_j, \ell_j)). \quad (1)$$

As we show in Figure 5, this notion of fairness allocates

more resources to “high-performing” users, i.e those who can generate large utility with a small amount of resources.

(ii) *Egalitarian welfare:* Here, we choose the allocation  $a$  which maximizes the egalitarian welfare (minimum of all utilities), i.e.  $a = \text{argmax } W_E$ , where

$$W_E = \min_{j \in \{1, \dots, n\}} u_j(a_j, \ell_j) = \min_{j \in \{1, \dots, n\}} u'_j(p_j(a_j, \ell_j)). \quad (2)$$

This allocates more resources to “struggling” jobs which need more resources to achieve large utility (Figure 5).

**Demand-based policies:** These policies apply when jobs have a well-defined SLO and it is possible to define its demand  $d_j$ . Such policies will compute allocations based on the demands of all jobs. This requires knowledge of the demand, which in turn depends on the performance mapping.

(iii) *No justified complaints (NJC) fairness [16, 17, 28]:* One class of demand-based policies which adopt the NJC fairness paradigm guarantee an equal share of  $R/n$  for each job. If the job’s demand is larger than  $R/n$ , it is allocated at least (but possibly more than) this share. But, if the job’s demand is smaller, the excess resources may be allocated to other jobs to improve overall resource usage. A user can have no justified complaints since they are either guaranteed to satisfy their SLOs or their utility will be larger than if they were to have  $R/n$  resources. To quantify this, we define the following metric. The term inside the minimum measures the utility achieved by job  $j$  with allocation  $a_j$  relative to the utility when using its fair share of  $R/n$  resources.

$$F_{\text{NJC}} = \min_{j \in \{1, \dots, n\}} \frac{u_j(a_j, \ell_j)}{u_j(R/n, \ell_j)} = \min_j \frac{u'_j(p_j(a_j, \ell_j))}{u'_j(p_j(R/n, \ell_j))} \quad (3)$$

In contrast to metrics such as the Jain’s index [31],  $F_{\text{NJC}}$  accounts for users’ performance when evaluating fairness. This metric has a maximum value of 1. Below, we describe a demand-based policy [16] which achieves  $F_{\text{NJC}} = 1$  while also using the resources efficiently as also shown in Figure 5.

*An NJC policy:* This policy proceeds iteratively. In the first round, it sets each user’s “share” to be  $R/n$ . It allocates  $d_j$  to each user  $j$  for whom  $d_j$  is smaller than the share. If  $n'$

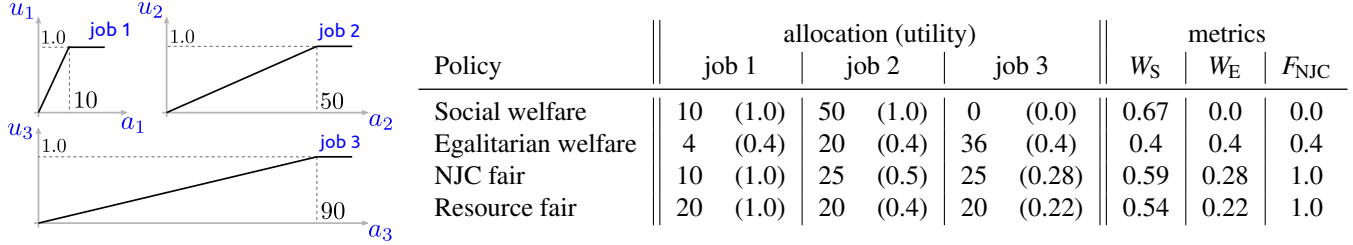


Figure 5: Comparison of the three (oracular) fair allocation criteria described in §4.1.1 in a synthetic example with 60 CPUs. *Left*: Utility curves for three jobs. The y axis is the utility and the x-axis is the number of resources. For simplicity, we have ignored the loads and assumed that utilities increase linearly up to the demand. The total demand is 150, whereas only 60 resources are available. *Right*: The allocations and utilities for each job under the three criteria. We have also shown the  $W_S$  (1),  $W_E$  (2), and  $F_{NJC}$  (3) metrics for each policy.

users were allocated  $R'$  resources in the first round, in the second round it sets each user's share to be  $(R - R')/(n - n')$ . It repeats this until all the remaining users' demands are larger than their share. It then divides up the remaining resources equally among the remaining users. While this policy may not maximize any welfare, it achieves Pareto-efficient user utilities. Another advantage of this policy is that it is strategy-proof, i.e. a user does not gain additional utility by falsely stating their demand [24, 28, 36].

This concludes our review of multi-tenant resource allocation objectives when performance mappings are known. We mention that prior work have used these objectives in various contexts with custom utilities. For instance, social welfare has been used in stream processing [35] and wireless networks [55], egalitarian welfare in video streaming [45], and several NJC policies are implemented in Mesos [29].

#### 4.1.2 Online learning policies in Cilantro

We will now develop our online policies. Our policies will operate on lower and upper confidence bounds obtained from the load forecasters and performance learners instead of the direct estimates; doing so accounts for the uncertainty in the learned models and encourages a policy to conservatively explore the space of allocations until the estimates become accurate. Cilantro's policies will proceed sequentially in allocation rounds. On round  $r$ , Cilantro chooses an allocation  $a^{(r)} = (a_1^{(r)}, \dots, a_n^{(r)})$  based on the feedback from all jobs up to now and the specific scheduling objective.

**Welfare-based online policies:** For welfare-based policies, Cilantro adopts the optimism in the face of uncertainty (OFU) principle [7]. OFU stipulates that, to maximize an uncertain function, we should choose actions which maximize an upper confidence bound (UCB) on the function. Both theoretically and empirically, OFU is known to outperform other strategies which use direct estimates or those which are pessimistic (i.e. maximize lower confidence bound). An in-depth exploration of OFU is beyond the scope of this work, but we refer the reader to relevant literature (e.g. [6, 8, 25, 53]).

While OFU is a well established design paradigm, most OFU policies are designed for end-to-end systems which output

a single reward signal. Adapting OFU for general welfare-based policies requires studying how the uncertainty in the performance and load translate to a UCB  $\hat{W}$  on the welfare  $W$  which we wish to maximize. Since  $W$  is non-decreasing in the utilities  $u_j$ , we can obtain a UCB for  $W$  by plugging in UCBs  $\hat{u}_j$  for the utility  $u_j$ , i.e.  $\hat{W} = W(\hat{u}_1, \dots, \hat{u}_n)$ . Similarly, since  $u_j$  is non-decreasing in the performance we can obtain a UCB by plugging in a UCB  $\hat{p}_j$  for  $p_j$ , i.e.  $\hat{u}_j = u'_j(\hat{p}_j)$ . This leads to the following choice of allocation on round  $r$ .

$$a^{(r)} = \operatorname{argmax}_{a \in \mathcal{A}^{(r)}} W \left( u'_1(p_1(a_1, \hat{\ell}_1)), \dots, u'_n(p_n(a_n, \hat{\ell}_n)) \right) \quad (4)$$

Above, since the exact load cannot be known, we conservatively over-estimate it via a UCB  $\hat{\ell}_j$  on the load. Here,  $\mathcal{A}^{(r)}$  is the allocation space on round  $r$  which is defined by two constraints: first, the total allocation cannot be larger than  $R$ , i.e.  $\sum_j a_j \leq R$ ; second, the current allocation cannot deviate too much from the previous allocation, i.e.  $a_j^{(r-1)} - B \leq a_j \leq a_j^{(r-1)} + B$  for all  $j$ , where  $B$  is a parameter to be specified. We impose the second constraint since large changes to allocations can have unpredictable effects on a job's performance; moreover, they take a long time to actuate, resulting in unreliable feedback while resources are being scaled up/down.

To optimize (4), one can use any off-the-shelf optimizer such as evolutionary algorithms, hill climbing, or integer programming which can handle the linear constraints for  $\mathcal{A}^{(r)}$ . In our implementation, we used an evolutionary algorithm (details in the appendix). Finally, we describe instantiations of this principle for the two welfare-based policies we saw in §4.1.1.

(i) Cilantro-SW: To emulate the social welfare policy in §4.1.1, on round  $r$ , we use the UCB for  $\hat{\ell}$  for load and  $\hat{p}$  for performance. Thus, we choose an allocation

$$a^{(r)} = \operatorname{argmax}_{(a_1, \dots, a_n) \in \mathcal{A}^{(r)}} \sum_{j=1}^n u'_j(\hat{p}_j(a_j, \hat{\ell}_j)).$$

(ii) Cilantro-EW: To emulate the egalitarian welfare policy in §4.1.1, on round  $r$ , we choose an allocation

$$a^{(r)} = \operatorname{argmax}_{(a_1, \dots, a_n) \in \mathcal{A}^{(r)}} \min_{j \in \{1, \dots, n\}} u'_j(\hat{p}_j(a_j, \hat{\ell}_j)),$$



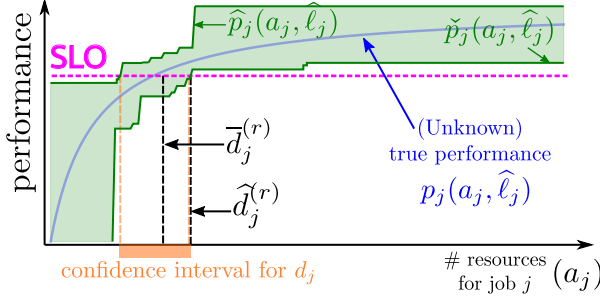


Figure 6: Illustration of Cilantro’s uncertainty-aware demand-based policies. We first obtain a UCB  $\hat{\ell}_j$  from the load forecaster, which ensures that we have a conservative estimate on the job’s load. In the figure, the  $x$  axis is the amount of resources  $a_j$  that could be allocated to job  $j$ . We show the SLO (pink), the slice of the unknown performance curve (blue) when the load is  $\hat{\ell}_j$ , and the confidence region obtained from past data (green). The LCB  $\check{p}_j$  and UCB  $\hat{p}_j$  on  $p_j(a, \hat{\ell}_j)$  are given by the lower and upper boundaries of the confidence region (solid green lines). A confidence interval for the demand (orange) can be obtained by the region where  $\hat{p}_j, \check{p}_j$  intersect the SLO line. To obtain a recommendation, we compute a UCB  $\tilde{d}_j^{(r)}$  on the demand (where SLO intersects  $\check{p}_j$ ) and  $\bar{d}_j^{(r)}$  via equation (5).

**Demand-based online policies:** For demand-based policies, on round  $r$ , we will use the confidence intervals from the performance learners and load forecasters to obtain conservative recommendations  $d_j^{(r)}$  for job  $j$ ’s demand. Then, we compute the allocations  $a^{(r)}$  for this round by invoking the same demand-based policy with the recommended demands  $\{d_1^{(r)}, \dots, d_n^{(r)}\}$  instead of the true demands.

Our method for obtaining demand recommendations is based on [36]. To describe this in more detail, observe that for demand-based policies it is sufficient to accurately estimate the demand well, i.e. it is not necessary to learn the entire performance mapping well. We have illustrated our strategy for obtaining the demand recommendation in Figure 6. First, we will denote by  $\tilde{d}_j^{(r)}$ , the UCB for the demand obtained as shown in Figure 6. As a conservative choice for this demand, we may wish to choose  $\bar{d}_j^{(r)}$  as the recommendation. However, we found that in practice this was overly conservative and the resulting allocations were very slow to adapt to feedback. Therefore, we also wish to use a more aggressive exploration strategy to reduce the uncertainty in our *demand*. We use:

$$\bar{d}_j^{(r)} = \operatorname{argmax}_{a_j} \min(\hat{p}_j(a_j, \hat{\ell}_j) - \text{SLO}, \text{SLO} - \check{p}_j(a_j, \hat{\ell}_j)) \quad (5)$$

To illustrate this rule, consider Figure 6 where  $\min(\hat{p}_j - \text{SLO}, \text{SLO} - \check{p}_j)$  is negative for large allocations when the performance LCB  $\check{p}_j$  is larger than the SLO and for small allocations where the performance UCB  $\hat{p}_j$  is smaller than the SLO. By maximizing (5), we are choosing points inside the confidence interval for the demand where both  $\check{p}_j, \hat{p}_j$  are further away from the SLO; so if job  $j$  were to receive  $\bar{d}_j^{(r)}$

resources, then we are most likely to reduce the demand uncertainty. However, choosing  $\bar{d}_j^{(r)}$  as the recommendation can lead to overly aggressive exploration so our final recommendation  $d_j^{(r)}$  is then obtained via,

$$d_j^{(r)} = \operatorname{clip}(\beta \tilde{d}_j^{(r)} + (1 - \beta) \bar{d}_j^{(r)}, d_j^{(r-1)} - B, d_j^{(r-1)} + B) \quad (6)$$

Here,  $\beta \in (0, 1)$  is a parameter to trade-off between  $\tilde{d}_j^{(r)}$  and  $\bar{d}_j^{(r)}$ . We clip this value between  $d_j^{(r-1)} - B$  and  $d_j^{(r-1)} + B$  to control wide deviations in resource allocations (similar to before). Next, we formally state Cilantro’s instantiation of the demand-based NJC procedure described in §4.1.1.

(iii) **Cilantro-NJC:** Here, we simply compute the recommended demand via (5), and then invoke the NJC procedure described in §4.1.1. In §7.1 we show that Cilantro-NJC retains some of the strategy-proofness properties of NJC.

## 4.2 Microservice resource allocation

Now, we will look another use-case for Cilantro, where we wish to optimize an end-to-end performance metric  $p$  of an application composed of several interdependent microservices (jobs). Examples for  $p$  include the total throughput of the application, the negative P99 latency, or even any combination of the two. Here, the entire fixed set of resources is available to the application and must be allocated between microservices for to maximize  $p$ . There are two main differences in this setting when compared to the multi-tenant setting which introduces new challenges. First, while assuming jobs run by different users are independent is reasonable when we aim to optimize for fairness, this is no longer true now since microservices within an application may have complex dependency graphs (see Figure 12-Left). Second, while an application’s performance is clearly tied to the performance of individual microservices, it is not possible to write it explicitly, as we did for the social or egalitarian welfare.

We overcome these challenges by modeling the end-to-end performance  $p$  as a direct function of the allocation to each microservice and the external load faced by the application. That is, we write  $p(a, \ell)$ , where  $a = (a_1, \dots, a_n)$  is a vector of allocations for each microservice and  $\ell$  is the external load on the application. On allocation round  $r$ , our online learning policy, which adopts the OFU principle, chooses an allocation vector which maximizes an upper confidence bound  $\hat{p}$  on the performance obtained from the performance learners:

$$a^{(r)} = \operatorname{argmax}_{a \in \mathcal{A}^{(r)}} \hat{p}(a, \ell). \quad (7)$$

While this circumvents accounting for individual microservice performance and dependency graphs, we now face the challenge of optimizing for an  $n$ -dimensional allocation with just one feedback signal. In contrast, in the multi-tenant setting we had more feedback (one for each job).

## 5 Discussion

We now present a discussion on Cilantro’s operation under various adversarial conditions that may occur in deployment.

**When online job feedback is unavailable.** Cilantro provides three fallback options when online feedback is not available. First, Cilantro allows a user to use a profiled model (using historical data) instead of online feedback. Second, it allows using proxy metrics from the Kubernetes API instead of real-world performance. In such cases, a user should specify how these proxies are tied to their utility and/or demand. Third, if neither of these is possible, we allow the user to directly submit an estimate for their resource demand which will then be fed to the policy when determining allocations. In such cases, we assume that utility increases linearly up to the demand when computing allocations. We evaluate this fallback option in §7.3. Due to Cilantro’s decoupled design, these fallback options can be effected with simple modifications to a job’s performance learner.

**Learning in unpredictable environments.** Some situations, such as unexpected load spikes for web services or interference between jobs, are fundamentally hard to predict. Cilantro’s uncertainty-aware design provides a degree of resilience against these unpredictable changes, as we show in its robustness to noise in load and resource demand estimates in Section 7.3. However, continued extreme fluctuations in the loads can negatively impact Cilantro’s performance. To avoid hysteresis when reallocating resources, future work can explore averaging loads over dynamically sized windows or including rules to temporarily override Cilantro’s policy.

**Limitations and Future Work.** Cilantro currently supports allocating only a single resource type. In our current implementation, multiple resource types can be bundled into grouping units, such as VM SKUs with a fixed ratio of CPU, Memory and GPUs, which can then be scheduled by Cilantro. However, such bundling is not always possible, especially when different jobs have different resource requirements. Extending Cilantro to handle multiple resource types is possible for welfare-based policies. However, learning and optimization can be challenging since the search space is now very large. Another related limitation is that Cilantro cannot handle non-fungible resource types. Cilantro also does not support online learning versions of market-based resource allocation policies in the multi-tenant setting [39, 56, 63]. These are avenues for future work to improve Cilantro. Cilantro also assumes utilities increase with increasing resources, however some workloads may demonstrate inverse scaling, especially when allocated resources become fragmented across physical nodes. Future work can relax this assumption by applying learning techniques robust to non-convex utility shapes. We also note that Cilantro can support multiple SLO parameters (e.g., for an inference job, ensuring a minimum latency and accuracy) by wrapping them in a single utility function, and

the design of such utility functions can be explored by future work.

## 6 Implementation

The Cilantro scheduler is implemented in 7600 lines of Python code, as a standalone scheduler for Kubernetes. Resource reallocation events are triggered by a timer-based event, which is raised every 2 minutes in our experiments. This window was chosen based on the fact that Kubernetes pods could be created and destroyed in 5-15 seconds. A 2 minute allocation round is long enough for the pod to reach its steady state that performance metrics from the job would be reliable, while at the same time frequent enough to adapt to changes in the load and learned performances.

To execute updated resource allocations received from policies, we horizontally scale the workloads by adding more replicas to their Kubernetes deployment. Newly created pods rely on the Kubernetes service discovery mechanism to connect to the workload’s other servers. The workload is responsible for load balancing queries onto the new servers. Workloads write logs to a volume shared with the sidecar cilantro client. The client parses performance metrics and then publishes them to the scheduler over gRPC. These messages also act as heartbeats to inform liveness to the scheduler.

The frequency of performance feedback depends on the application and the environment. For instance, database serving jobs may report feedback multiple times in a minute, while ML training jobs may do so once every few minutes. To avoid bottlenecks, we use an asynchronous design for Cilantro where each component operates in a push or pull based framework. This allows high frequency components to operate at their maximum rate while allowing slower components, such as learners for low-frequency jobs or cluster managers, to be polled when required.

**Specifying utilities and objectives.** Utilities of jobs are calculated based on the performance metrics collected by the Cilantro clients in the last resource allocation round. To compute the utilities, application developers specify utility as a python method which operates on a list of floating point numbers representing the performance metrics observed in the previous resource allocation round. Similarly, the scheduling objective (e.g., social welfare from §4.1.1) is also defined by the cluster operator as a python method operating on the list of utilities from all jobs.

**Learning models and load forecasters.** For the multi-tenant setting, we used a tree-based binning estimator [8, 27, 36] with Lipschitz constant 10 for each job’s resource-to-performance estimation. This is a simple and computationally efficient estimator, but does not work well in high dimensions. Therefore, for the microservices setting where we have a high dimensional estimation challenge, we use kernel ridge regression [59, 65] with a Matern kernel with smoothness parameter

set to 2.5. In both settings, for the load forecasters, we use an autoregressive moving average (ARMA) model [41] with autoregressive order 1 and moving average order 1. Finally, all confidence bounds were computed at the 90% level, meaning that the probability that the true parameter lies between the upper and lower confidence bounds is 90%. We used the above learning models since they are simple and have few tunable hyperparameters. With Cilantro’s modular design, these can be easily swapped with any other model as long as they provide reliable uncertainty estimates.

**Other policy parameters:** For all our policies, we set the parameter  $B$  which controls the deviation from the previous allocation to 10. For demand-based policies, we set the parameter  $\beta$  which trades off between conservative and aggressive exploration to  $3/4$ . For the welfare-based policies in §4.1 and the microservices use case in §4.2, we use evolutionary algorithms to optimize the UCBs. The exact implementation is described in the appendix.

## 7 Evaluation

We evaluate Cilantro in two settings described in §4.

1. In the multi-tenant setting, Cilantro’s online learning policies, which do not start with any prior data, are competitive with oracular policies which have access to jobs’ resource to performance mappings obtained after several hours of profiling. Moreover, they outperform 9 other baselines on the metrics outlined in §4.1.
2. In the microservices setting, Cilantro is able to support the completely different objective of minimizing end-to-end latency. It outperforms three other baselines and reduces the P99 latency to  $\times 0.57$  that achieved by the next best performance-aware baseline.
3. In our microbenchmarks, we show that Cilantro’s allocation policies are inexpensive, evaluate its fallback options when performance metrics are unavailable, and demonstrate its robustness to errors in feedback and choices for performance learner and forecaster models.

### 7.1 Multi-tenant cluster sharing

We first evaluate Cilantro’s multi-tenant policies (§ 4.1.2) on a 1000 CPU cluster shared by 20 users.

**Workloads.** We use three classes of workloads—database querying, prediction serving and machine learning training—which are used to create multiple jobs. The database querying workload runs TPC-DS [43] queries on replicated instances of sqlite3 database and uses the query latency as the performance metric. The prediction serving workload runs queries on a ML model (random forest regressor) trained on the news popularity dataset [20]. The ML training workload trains a neural network on the naval propulsion [12] dataset using stochastic gradient descent. The database querying and prediction serving workloads use the query latency as the performance

metric while ML training uses batch throughput to measure performance. Resource-performance mappings for informing the oracle baselines in §7.1 were obtained through offline profiling of all workloads. These profiles are visualized in Figure 7. More details of the workloads, including workload-specific parameters are available in the appendix.

**Traces.** Queries to the database and prediction serving workloads are dispatched by a trace-driven workload generator. We use the Twitter API [5] to collect a trace of tweet arrival rates at Twitter’s Asia datacenters; to bring to parity with our cluster, we subsample the arrival rate by a factor of 10. For the ML training workload, we draw queries from an essentially infinite pool to create a constant stream of work.

**Experimental set up.** We use a cluster of 250 AWS m5.xlarge instances (4 vCPUs each). The Cilantro scheduler runs on its own dedicated m5.xlarge instance. We use the above 4 workloads to create 20 jobs as follows: 10 database jobs with P90, P90, P90, P90, P95, P95, P95, P95, P99, P99 latency SLOs of 2s; 3 prediction serving jobs with P90, P90, and P95 latency SLOs of 2s; 7 ML training jobs with throughput SLOs of 400, 400, 450, 450, 500, 500, and 500 QPS. To reflect settings where small SLO violations may be either critical or inconsequential, we discount the utility via one of the three options in Fig. 4 for each job. Detailed information on the users’ jobs is given in the appendix. The estimated total amount of resources based on the median demand was 1637 CPUs; hence, even at full capacity, not all users can satisfy their SLOs. We evaluate all baselines for 6 hours.

#### 7.1.1 Baselines

**Oracular policies.** We implement the three policies in §4.1.1 with oracular access to the true performance mappings (obtained by exhaustively profiling workloads for at least 4 hours). They are Oracle-SW, Oracle-EW, for maximizing social/egalitarian welfare and the Oracle-NJC fairness policy.

**Cilantro policies.** We evaluate Cilantro-SW, Cilantro-EW, and Cilantro-NJC, as described in Sec. 4.1.2.

**Other heuristics.** We implement four methods for fairness and maximizing welfare. While not based directly off specific prior work, such methods are common in the scheduling literature [13,26]. Resource-Fair simply allocates an equal amount of resources to each job. EvoAlg-SW and EvoAlg-EW are evolutionary algorithms for social and egalitarian welfare; the same procedure used for Cilantro’s welfare policies, but now operating directly on the performance metrics. Greedy-EW starts by allocating resources equally; on each round, it evaluates job utilities in the previous round and takes away one CPU each from the top half of the users who had high utility and allocates it to the bottom half.

**Baselines from prior work.** We adapt five feedback-driven methods from prior work - Ernest [58], Quasar [15], Minerva [45], Parties [10] and MIAD (Multiplicative-

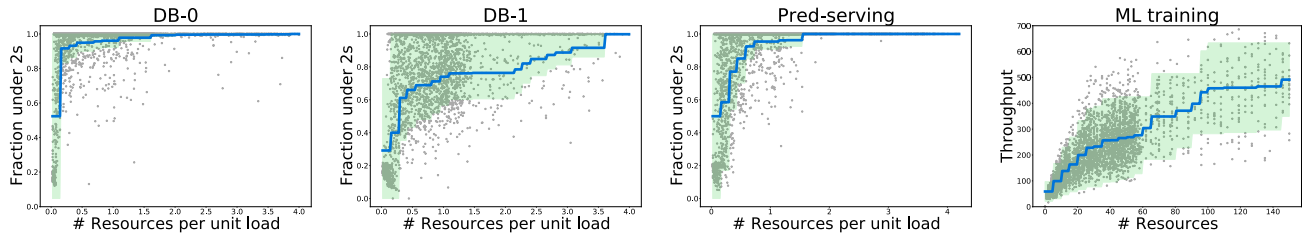


Figure 7: Performance vs resource-allocation-per-unit-load obtained after profiling the database querying, prediction serving and ML training workloads. The blue curve is the average performance value and the shaded region is the  $2\sigma$  confidence interval. For the latency-based workloads (DB-0, DB-1, and prediction serving), we show the number of resources per unit load (arrival QPS) on the  $x$ -axis and the fraction of queries completed under 2s on the  $y$ -axis. For the ML training workload, we show the number of resources on the  $x$  axis the amount of data processed per second on the  $y$ -axis. To obtain accurate estimates, we sampled low resources allocations more densely.

Increase/Additive-Decrease) [11]. In particular, we note that applying the Parties notion of migration in our setting would imply moving the job to a different cluster or increasing the size of the cluster, both of which are beyond scope for this fixed cluster setting. Details on the specific adaptations are available in the appendix.

### 7.1.2 Results & Discussion

**Evaluation on performance-aware fairness metrics.** We first compare all 15 baselines on the social welfare (1), egalitarian welfare (2), and the NJC fairness criteria (3). Fig. 8 illustrates the results by plotting the time-averaged NJC fairness vs the two welfare criteria. Table 2 (in the appendix) tabulates these values explicitly with error bars. While the oracular methods perform best on their respective metrics, we find that the online learning policies in Cilantro come close to matching them. Resource-Fair achieves a perfect NJC score by definition, but performs poorly on social and egalitarian welfare as it is performance oblivious.

We found that Greedy-EW, Parties, and MIAD were sensitive to the amount by which we changed the allocations based on feedback; when tuning them, we found that they were either too slow or too aggressive when responding to load shifts. Next, the learning models used by Quasar and Ernest were not able to accurately estimate the demands in our experiment. Finally, the evolutionary baselines were inefficient, taking a long time to discover the optimal solution. They, however, were effective within Cilantro’s welfare policies when you need to optimize a cheap analytically computable function as they can be run for several iterations.

Despite our general approach, Cilantro’s policies are able to outperform Minerva and Greedy-EW which are designed specifically to maximize egalitarian welfare. It also outperforms generically designed evolutionary algorithms for the social and egalitarian welfare. While it may indeed be possible to design more efficient fine-tuned policies for a given objective, the flexibility provided by Cilantro’s approach is beneficial to end users. It should not be surprising that Cilantro outperforms other systems such as Ernest, Quasar, Parties, and MIAD as our policies are designed to explicitly optimize

for these objectives. *But this is precisely the goal of Cilantro.* End-users can declare their desired objective, and Cilantro will automatically derive policies to achieve them.

To illustrate how Cilantro improves with feedback, in Fig. 9, we have shown how the three objectives evolve over time for Cilantro’s policies. Resource-Fair trivially achieves  $F_{\text{NJC}} = 1$  at start since our initial allocation is always 50 CPUs to each job (i.e Resource-Fair). However, it does poorly on welfare due to poor cluster usage. The goal behind Cilantro-NJC is to achieve  $F_{\text{NJC}} = 1$  while also achieving good cluster usage. This causes the initial drop in performance for Cilantro-NJC as it explores better allocations that still maximize  $F_{\text{NJC}}$ .

Table 2 presents the detailed results of our multi-tenant cluster resource sharing evaluation. This table adds a metric which measures the useful resource usage.

$$\text{Useful resource usage} = \sum_{j=1}^m \min(a_j, d_j) \quad (8)$$

Here, the  $d_j$  is user  $j$ ’s resource demand. This demand-based metric, measures how much *useful* work is being done by the cluster as allocations beyond the demand do not increase a user’s utility (see Fig. 4). We find that Cilantro’s policies achieve the maximum useful resource usage in their respective classes. This is because learning resource demands allows Cilantro to reallocate resources from jobs which have already achieved maximum utility to jobs which can benefit from increased resources.

**Individual user utilities.** To delve deeper into the trade-offs of the three paradigms discussed in §4.1, we have shown the individual user utilities achieved by these three policies in Fig. 10. We see that both the social and egalitarian welfare policies result in some users being worse off than receiving their fair allocation of  $1000/20 = 50$  CPUs. This results in an NJC fairness violation. In contrast, in Cilantro-NJC, users are at most marginally worse off than their fair share. However, a third of the users achieve a noticeably higher utility than their fair share utility, with more than  $3\times$  for a few of them. We also see that Cilantro-EW has maximized egalitarian welfare by taking resources away from those who achieve high utility and giving it to those who do not, while Cilantro-SW has



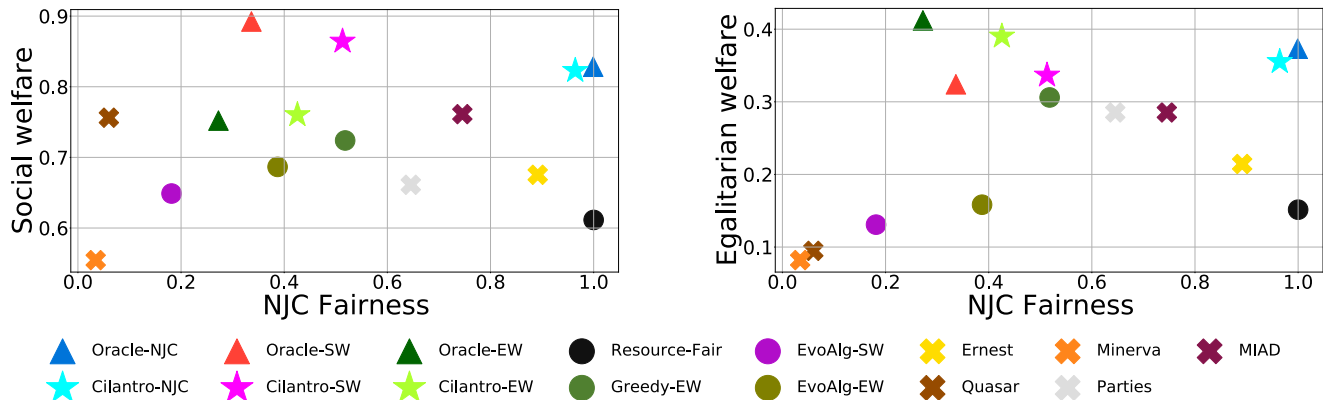


Figure 8: NJC fairness vs the social and egalitarian welfare (see §4.1.1) for all policies. We report the average value over the 6 hour period. Higher is better for all metrics, so closer to the top right corner is desirable. The Oracle-SW, Oracle-EW policies optimize for the social and egalitarian welfare when the performance mappings are known and Oracle-NJC achieves maximum fairness while improving cluster usage. The corresponding Cilantro policies are designed to do the same without a priori knowledge of the performance mappings.

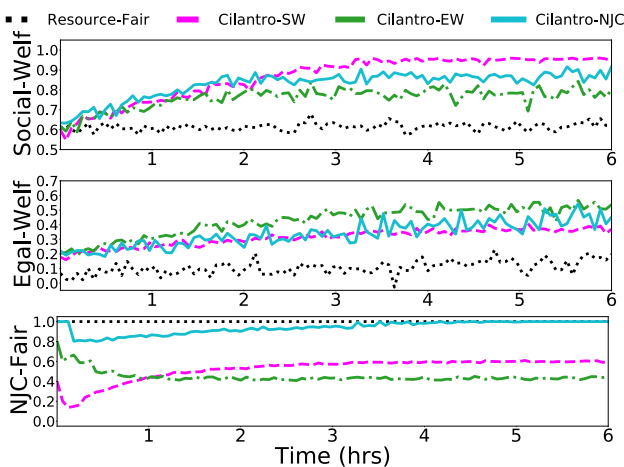


Figure 9: Convergence over time of social, egalitarian welfares and NJC fairness for the three Cilantro policies.

maximized social welfare by allocating more resources to jobs that can quickly achieve high utility.

**Evaluating Strategy-proofness.** We next evaluate Cilantro policies for strategy-proofness. A policy is said to be strategy-proof if an unscrupulous user cannot increase the utility of their job by misreporting their performance metrics to the scheduling policy. For this, we repeat the same experiment set up; all jobs behave exactly as before except the db16 job which lies about its performance by either under-reporting by a factor  $\times 1/2$ , or over-reporting by a factor  $\times 2$ . By under-reporting, the user gives the impression that more resources are required to reach its SLO; in contrast, by over-reporting, a user is deceiving the scheduler to prioritize their job as they can achieve high utility with few resources. In Fig. 11, we report the utilities achieved by db16 under these untruthful behaviors. We see that for Cilantro-NJC, the job’s utility does not increase when over-reporting and decreases when under-reporting, leaving no incentive for the user to be untruthful.

In contrast, for Cilantro-EW, a user stands to gain by under-reporting while for Cilantro-SW, they gain by over-reporting. While a theoretical study of such strategy-proofness properties is beyond the scope of this work, it is interesting to empirically observe that the strategy-proofness properties of NJC fairness policies are retained in Cilantro.

## 7.2 Resource allocation for Microservices

We now demonstrate the use of Cilantro to allocate resources for inter-dependent microservices serving an application. A query to the application triggers multiple queries to different microservices and the final result is returned to the user. Cilantro must observe a single end-to-end metric, the end-to-end query latency, and then allocate fixed cluster resources to different microservices to minimize the P99 latency of the application. We note that Cilantro does not require meta information about the microservices, such as their dependency and control flow graphs; Cilantro directly optimizes the end-to-end metric as described in §7.2.

**Workload.** We use the Hotel Reservation application from DeathStarBench [22]. It has 19 microservices, including 6 MongoDB databases, 3 memcached kv-stores and a nginx webserver running on a consul service mesh. The architecture is shown in Fig. 12-Left. Collectively, these microservices serve search, recommendation, rating, account management and geolocation queries from users. We use wrk2 [54] to process and submit the query workload provided in [22]. We measure the end-to-end latency of queries submitted to the frontend microservice. All microservices experiments are run on a 160 CPU cluster with 20 AWS m5.2xlarge instances.

**Baselines.** We compare Cilantro’s end-to-end policy (§4.2) against three baselines. Resource-Fair always equally allocates the resources among microservices. EvoAlg is an evolutionary algorithm which optimizes for the P99 latency. e-greedy randomly picks a new allocation with probability  $1/3$ ,

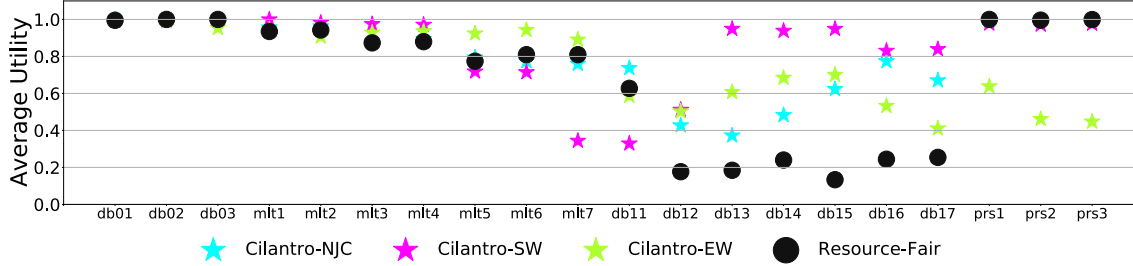


Figure 10: The average utility achieved by the 20 jobs for the three online learning methods in Cilantro and Resource-Fair. Here, db0x, mltx, db1x, and prsx refers to jobs using the DB-0, ML training, DB-1, and prediction serving workloads from § 7.1.

Policy	Social Welfare, ( $W_S$ )	Egalitarian Welfare ( $W_E$ )	NJC Fairness ( $F_{NJC}$ )	Useful resource usage
Oracle-SW	$0.892 \pm 0.004$	$0.324 \pm 0.008$	$0.336 \pm 0.004$	$0.964 \pm 0.002$
Oracle-EW	$0.752 \pm 0.003$	$0.412 \pm 0.007$	$0.272 \pm 0.002$	$0.997 \pm 0.000$
Oracle-NJC	$0.828 \pm 0.002$	$0.373 \pm 0.008$	$0.999 \pm 0.000$	$0.991 \pm 0.000$
Cilantro-SW	<b><math>0.864 \pm 0.006</math></b>	$0.337 \pm 0.013$	$0.513 \pm 0.020$	$0.818 \pm 0.012$
Cilantro-EW	$0.760 \pm 0.007$	<b><math>0.390 \pm 0.020</math></b>	$0.426 \pm 0.037$	<b><math>0.954 \pm 0.012</math></b>
Cilantro-NJC	$0.823 \pm 0.002$	$0.355 \pm 0.005$	<b><math>0.964 \pm 0.006</math></b>	$0.931 \pm 0.003$
EvoAlg-SW	$0.649 \pm 0.017$	$0.131 \pm 0.016$	$0.182 \pm 0.048$	$0.671 \pm 0.021$
EvoAlg-EW	$0.687 \pm 0.011$	$0.158 \pm 0.012$	$0.387 \pm 0.040$	$0.700 \pm 0.009$
Resource-Fair	$0.611 \pm 0.002$	$0.151 \pm 0.006$	<b><math>1.000 \pm 0.000</math></b>	$0.766 \pm 0.001$
Greedy-EW	$0.724 \pm 0.005$	$0.306 \pm 0.006$	$0.518 \pm 0.009$	$0.882 \pm 0.004$
Ernest	$0.675 \pm 0.002$	$0.214 \pm 0.005$	$0.891 \pm 0.013$	$0.774 \pm 0.002$
Quasar	$0.756 \pm 0.002$	$0.095 \pm 0.003$	$0.060 \pm 0.003$	$0.706 \pm 0.002$
Minerva	$0.555 \pm 0.017$	$0.082 \pm 0.006$	$0.034 \pm 0.005$	$0.407 \pm 0.023$
Parties	$0.661 \pm 0.002$	$0.285 \pm 0.006$	$0.645 \pm 0.000$	$0.766 \pm 0.001$
MIAD	$0.761 \pm 0.002$	$0.285 \pm 0.005$	$0.745 \pm 0.000$	$0.766 \pm 0.001$

Table 2: The social welfare (1), egalitarian welfare (2), NJC fairness metric (3), and the effective resource usage (8) for all 13 methods. Higher is better for all four metrics, and the maximum and minimum possible values for all metrics are 1 and 0. The values shown in bold have achieved the highest value for the specific metric, besides the oracular policies. Resource-Fair has NJC fairness  $F_{NJC} = 1$  by definition.

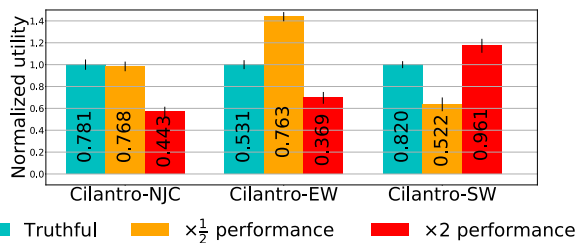


Figure 11: The utility of db16 under the three online learning policies, when they report truthfully, when they under-report, and when they over-report. The plot normalizes with respect to truthful reporting, but the bars are annotated with the absolute value.

or uses the allocation with the smallest observed P99 latency with probability 2/3.

**Results and Discussion.** Fig. 12 shows how the instantaneous and time-averaged P99 latency (computed in 30s intervals) evolves with time during the course of the experiment. Both Cilantro and EvoAlg explore early on (Fig. 12-Center), but as they find better values, exploration shrinks as they focus

on testing more promising allocations. However, Cilantro’s OFU-based online learning policy is able to do this more effectively than EvoAlg.  $\epsilon$ -greedy explores aggressively even in later stages and is unable to adequately exploit good candidates it may have discovered in the early stages. Overall, Cilantro achieves a mean P99 of 525ms, compared to 930ms for EvoAlg, the next best baseline.

### 7.3 Microbenchmarks

**Cilantro Overhead.** Fig. 13-Left evaluates the time taken for Cilantro to process the feedback and compute the allocations for the three policies described in Sec. 4.1. This shows that Cilantro is fairly light-weight. For comparison, the average time it took to de-allocate a Kubernetes pod and assign it to a different job was on the order of 5-15s.

**Unavailable performance metrics.** In real-world situations, performance metrics of all users may not be available. We evaluate Cilantro’s fallback defaults for such instances. We re-run the same experiment in § 7.1, but for users db01, ml1, and

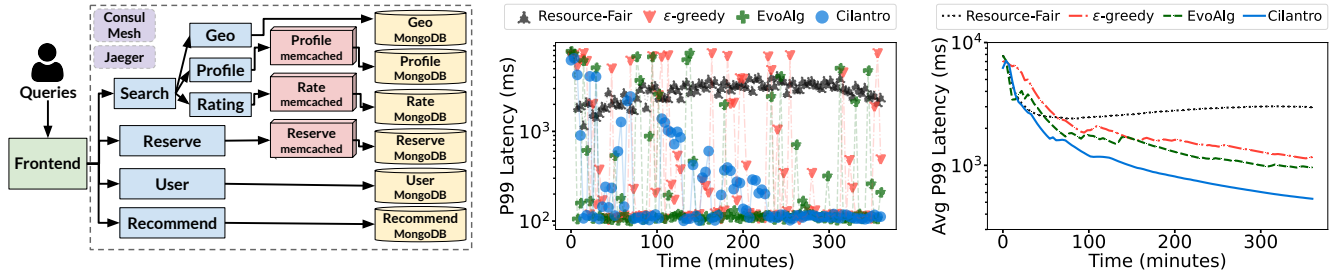


Figure 12: *Left*: Microservices architecture of the hotel reservation benchmark [22]. Blue boxes are business logic, red boxes are caching services, yellow boxes are databases and purple boxes are networking services. *Center*: Results for the microservices experiment comparing four methods on P99 latency over 6 hours, plotting the instantaneous P99 latency vs time. *Right*: The time-averaged P99 latency vs time.

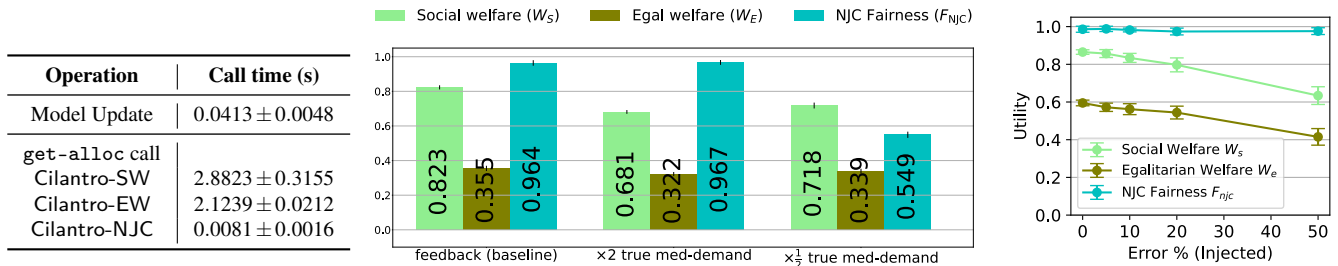


Figure 13: Cilantro microbenchmarks. *Left*: Mean time taken (in seconds) by Cilantro to update the performance model and for computing a new allocation for each of the three fixed cluster sharing policies. *Center*: Evaluation of Cilantro’s fallback option, where users provide a demand value if they cannot report performance metrics. We evaluate Cilantro-NJC when 5 out of 20 users use this option. Since the true demand cannot be known, we use either half or twice the true demand under the median load from our profiled data. *Right*: The three performance metrics for Cilantro-NJC when we artificially introduce error to the confidence intervals of the performance and load.

mlt2, db11, and prs1, we manually set the demand as described in §5. Since the true demands are not known a priori, users might under- or overstate them. To reflect this, we first compute the true demand for each user under the median load from our profiled data. We evaluate Cilantro-NJC when these five users report either half this value as their demand or twice this value, when compared to providing feedback. Fig. 13-Center presents results on the three criteria given in §4.1. While the fallback options are worse than when reporting feedback, the failures are graceful. Cilantro is still able to learn from the remaining 15 users and achieve efficient allocations with only relatively small drops in social and egalitarian welfare. The NJC fairness criterion is significantly small when under-reporting since these 5 users will have been allocated at most half of their true demand and  $F_{\text{NJC}}$  (3) depends on the single worst fairness violation.

#### Robustness to choice of learners and feedback errors.

While Cilantro’s decoupled design aids with generality, it may be susceptible to the idiosyncrasies of the specific models used for the performance learners and load forecasters. Moreover, in many real environments, the feedback can be very noisy. To show that Cilantro is robust to both these effects, we perform the following microbenchmark in a synthetic 5 user environment (described in the Appendix) with the Cilantro-NJC policy. As both feedback noise and model idiosyncrasies can be modeled with inaccurate confidence intervals, we introduce increasing levels of noise (5%, 10%,

20%, 50%) to the upper and lower confidence bounds returned by the learners and forecasters. The results, given in Fig. 13-Right, show that the social and egalitarian welfare decrease gracefully with noise. Moreover, due to Cilantro-NJC’s conservative approach for demand recommendations, the NJC fairness metric remains relatively high despite the noise.

## 8 Conclusion

We described Cilantro, a performance-aware framework for the allocation of a finite amount of resources among competing jobs. Our motivations were: (i) resource allocation policies should be performance-aware and based on real-time feedback in production environments, (ii) schedulers should accommodate diverse allocation objectives. We designed Cilantro to address these challenges by decoupling the performance learning from the policies and informing the policies of uncertainties in performance estimates, thus enabling the realization of several performance-aware policies in multi-tenant and microservices settings.

## 9 Acknowledgements

We thank the OSDI reviewers and our shepherd, Tim Harris, for their invaluable feedback. This work is in part supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

## References

- [1] Amazon Compute Service Level Agreement. <https://aws.amazon.com/compute/sla/>, 2022.
- [2] Hadoop Fair Scheduler. <https://hadoop.apache.org/>, 2022.
- [3] Kubernetes api health endpoints | kubernetes. <https://kubernetes.io/docs/reference/using-api/health-checks/>, 2022.
- [4] Ray dashboard — ray v1.7.0. <https://docs.ray.io/en/latest/ray-dashboard.html>, 2022.
- [5] Twitter Streaming API. <https://developer.twitter.com>, 2022.
- [6] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.
- [7] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721*, 2012.
- [8] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvari. X-armed Bandits. *arXiv preprint arXiv:1001.4475*, 2010.
- [9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.
- [10] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.
- [11] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.
- [12] Andrea Coraddu, Luca Oneto, Alessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. Machine learning approaches for improving condition-based maintenance of naval propulsion plants. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 230(1):136–153, 2016.
- [13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.
- [14] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.
- [15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
- [16] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.
- [17] Danny Dolev, Dror G Feitelson, Joseph Y Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: On fair sharing of multiple resources. In *proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 68–75, 2012.
- [18] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys ’12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.
- [19] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys ’12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.
- [20] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. A proactive intelligent decision support system for predicting the popularity of online news. In *Portuguese Conference on Artificial Intelligence*, 2015.
- [21] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.
- [22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvisky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
- [23] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 conference on*



*Applications, technologies, architectures, and protocols for computer communication*, pages 1–12, 2012.

- [24] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.
- [25] Alkis Gotovos. Active learning for level set estimation. Master’s thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science., 2013.
- [26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 65–80, 2016.
- [27] Jean-Bastien Grill, Michal Valko, and Rémi Munos. Black-box optimization of noisy functions with unknown smoothness. In *Advances in Neural Information Processing Systems*, pages 667–675, 2015.
- [28] Avital Gutman and Noam Nisan. Fair allocation without trade. *arXiv preprint arXiv:1204.4286*, 2012.
- [29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
- [30] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.
- [31] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.
- [32] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.
- [33] Sangeetha Abdu Jyothi, Carlo Curino, Isha Menache, Shравan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, November 2016. USENIX Association.
- [34] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.
- [35] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’18, page 249–262, New York, NY, USA, 2018. Association for Computing Machinery.
- [36] Kirthevasan Kandasamy, Gur-Eyal Sela, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Online learning demands in max-min fairness. *arXiv preprint arXiv:2012.08648*, 2020.
- [37] Mamoru Kaneko and Kenjiro Nakamura. The nash social welfare function. *Econometrica: Journal of the Econometric Society*, pages 423–435, 1979.
- [38] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.
- [39] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.
- [40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [41] Spyros Makridakis and Michele Hibon. Arma models and the box-jenkins methodology. *Journal of forecasting*, 16(3):147–163, 1997.
- [42] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on networking*, 8(5):556–567, 2000.
- [43] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB ’06*, page 1049–1058. VLDB Endowment, 2006.
- [44] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

- [45] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.
- [46] Hiep Chi Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *International Conference on Automation and Computing*, 2013.
- [47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.
- [48] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29(5):79–95, dec 1995.
- [49] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.
- [50] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12*, New York, NY, USA, 2012. Association for Computing Machinery.
- [51] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmerek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [52] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.
- [53] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.
- [54] Gil Tene. giltene/wrk2: A constant throughput, correct latency recording variant of wrk. <https://github.com/giltene/wrk2>. (Accessed on 04/19/2022).
- [55] Yan Kyaw Tun, Nguyen H Tran, Duy Trong Ngo, Shashi Raj Pandey, Zhu Han, and Choong Seon Hong. Wireless network slicing: Generalized kelly mechanism-based resource allocation. *IEEE Journal on Selected Areas in Communications*, 37(8):1794–1807, 2019.
- [56] Hal R Varian. Equity, envy, and efficiency. 1973.
- [57] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.
- [58] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.
- [59] Max Welling. Kernel ridge regression. *Max Welling's classnotes in machine learning*, pages 1–3, 2013.
- [60] John Wilkes. *Utility Functions, Prices, and Negotiation*, chapter 4, pages 67–88. John Wiley and Sons, Ltd, 2009.
- [61] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.
- [62] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.
- [63] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C Lee. Amdahl's law in the datacenter era: A market for fair processor allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2018.
- [64] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: MI-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.
- [65] Yuchen Zhang, John Duchi, and Martin Wainwright. Divide and conquer kernel ridge regression. In *Conference on learning theory*, pages 592–617. PMLR, 2013.

- [66] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

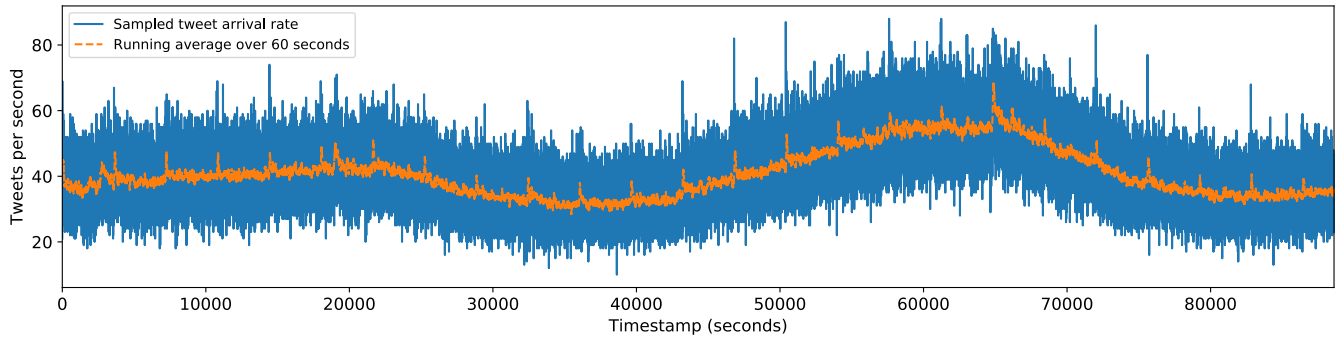


Figure 14: Sampled query arrival rate from the twitter trace collected over the duration of a day.

## A Experiment Addendum

### A.1 Workload description

**Database querying:** We use the TPC-DS [43] benchmark suite as the workload backed by replicated instances of sqlite3 database. From the TPC-DS query set, we created two workloads (setting scale factor to 100): DB-0, which had queries that completed in under 100 ms and DB-1 which had queries that had a completion time between 100 and 300 ms. When a query is requested, we randomly pick a relevant query and dispatch it according to the trace. The performance metric of interest is query latency.

**Prediction serving:** In prediction serving [13], a job processes arriving queries to output a prediction, usually obtained via a machine learning model. In our set up, we use a random forest regressor as the model and the the news popularity dataset [20] for training and test queries in a 50:50 split. Queries are picked randomly from the test set and issued in batches of 4. The metric of interest is the serving latency.

**ML training:** We use CPUs to train a neural network with four hidden layers of size 64 each. We train our model on the naval propulsion [12] dataset using stochastic gradient descent (SGD). Each task in this workload consists of training a batch of 16 points for 100 iterations. The performance metric of interest here is the batch throughput.

### A.2 Environment details

**Workload traces.** As described in §7.1, we use traces collected from twitter to generate traffic patterns for our workloads. The query arrival rate of this trace is visualized in Figure 14.

**Multi-tenant cluster jobs setup.** For the multi-tenant cluster resource sharing evaluation, we setup 20 jobs with different workloads and SLOs as described in in §7.1. Table 3 details the exact SLO and utility function for each job. The utility function for each job is either of `linear`, which directly maps performance to utility (Figure 4(a)), `sqrt`, which performs a sublinear mapping of performance to utility (Figure 4(b)), or `quadratic`, which performs a superlinear mapping of performance to utility (Figure 4(c)).

**TPC-DS Query Binning.** The queries used for the db serving workload in §7.1 were selected from the TPC-DS benchmark suite. The TPC-DS suite consists of 99 query templates out of which 27 were not compatible with the sqlite dialect and were discarded. The remainder were binned according to their mean latency when measured on a AWS m5.2xlarge instance. The chosen query types and their ids are listed in Table 4

### A.3 Baselines from prior work

Here we describe the specific implementation of prior work baselines used in Section 7.

- 1) Ernest [58]: Ernest uses a featurized linear model to estimate the time taken to run a job. We use this estimate to approximate the resource demand to meet the job’s SLO. On each round, we use the estimated demand as inputs to NJC to compute the allocations.
- 2) Quasar [15]: Quasar uses collaborative filtering to estimate a job’s resource demand, which we use as inputs to NJC to compute the allocations. We do not incorporate mechanisms for vertical scaling and workload co-location described in [15] to be consistent across all methods.



Job and SLO Type	Job Name	SLO	Utility Function
Database Serving (Latency)	db01	0.9	linear
	db02	0.9	linear
	db03	0.95	sqrt
	db11	0.9	linear
	db12	0.9	quadratic
	db13	0.95	quadratic
	db14	0.95	linear
	db15	0.95	quadratic
	db16	0.99	quadratic
Prediction Serving (Latency)	prs1	0.9	linear
	prs2	0.9	sqrt
	prs3	0.95	sqrt
ML Training (Throughput)	mlt1	400	sqrt
	mlt2	400	sqrt
	mlt3	450	linear
	mlt4	450	linear
	mlt5	500	quadratic
	mlt6	500	quadratic
	mlt7	500	quadratic

Table 3: SLO and utility functions used for jobs in experiments in §7.1. For Latency based SLOs, the SLO implies the fraction of queries that completed under 2 seconds. For Throughput based SLOs, the SLO is the desired query rate, measured in queries per second.

Query Bin	TPC-DS Query Ids	Mean Execution Time (s)
db0	93, 91, 92, 45, 85, 15, 32	0.28
db1	90, 84, 8, 55, 96, 81, 79	0.67

Table 4: Details of the bins created from TPC-DS queries. Each user’s workload is generated using these bins. Execution time is profiled on a SQLite3 database running on AWS m5.2xlarge instance with one allocated CPU core.

- 3) Minerva [45]: Minerva sets the allocation for job  $j$  at each step to be proportional to  $a_j/u_j$  where  $a_j$  and  $u_j$  are the allocation and utility at the previous round.
- 4) Parties [10]: Parties upsizes the allocation for a job if it violates or is close to violating the SLO, downsizes the allocation if the job comfortably satisfies the SLO, and otherwise does nothing. If the SLOs of all jobs cannot be met, it evicts the job from the server. As eviction is not an option in our setting we use the Parties logic to compute the demands which are then fed to NJC to obtain the allocations. For upsizing, we increase the demand by 20 CPUs and for downsizing, we decrease it by 5. These parameters were tuned so that the policy did reasonably well on all three metrics.
- 5) MIAD (Multiplicative-Increase/Additive-Decrease) [11]: This is inspired by TCP congestion control. If a user’s job violates the SLO, we increase its demand by  $1.5 \times$  the current allocation, and if it satisfies the SLO, we set the demand to be one minus the current allocation. We then invoke NJC to compute the allocation for the next round. These parameters were tuned so that the policy did reasonably well on all three metrics.

## A.4 Evolutionary Algorithm

We describe the evolutionary algorithm used in all of our experiments, i.e. to optimize the profiled information for the oracular welfare policies, to optimize the upper confidence bounds for the learning policies in §4.1.2 and §4.2, and the evolutionary algorithm baselines in §7.1 and §7.2. The input to the algorithm is a data source which the algorithm can query using an allocation and obtain a feedback signal. This data source can either be a cheap analytically computable function available in memory, as is the case for the oracles and learning policies, or an expensive experiment, as is the case when used as a baseline to directly

optimize for performance. The algorithm maintains a hash table mapping allocations to mean observed signal values. When it receives feedback for an allocation, it updates the mean value if the allocation has already been tried, or it creates a new entry and stores the feedback.

Our evolutionary algorithm proceeds as follows. It has an initialization phase of 10 rounds. In the first 2 rounds, it always queries a resource-fair allocation. In the remaining 8 rounds, it queries a random allocation  $a$  such that  $\sum_{j=1}^n a_j = R$ . On each subsequent round, it chooses a random allocation in the above manner with probability 0.1. With probability 0.9, it samples one of the existing allocations in the hash table based on the mean feedback value, performs a mutation operation, and queries the new allocation obtained via the mutation. We now describe these two steps.

- *Sampling*: Let  $\{(a_i, y_i)\}_i$  be the (allocation, mean feedback) pairs in the hash table. Let  $m, s$  denote the mean and standard deviation of the  $\{y_i\}$  values. We sample  $a_i$  with probability proportional to  $\exp((y_i - m)/s)$ .
- *Mutation*: The mutation operation is composed of a sequence of steps to modify a given allocation  $a$ . At each step, we randomly sample one job  $j$  which has an allocation of at least 2 CPUs; we then sample any other job  $k \neq j$ ; we then decrease  $j$ 's allocation by 1 and increase  $k$ 's allocation by 1. The number of steps is chosen uniformly at random between 1 and 20.

## A.5 Other experimental details

**Synthetic environment for robustness microbenchmark:** For the microbenchmark in Fig. 13(left), we use 5 users whose load is obtained by the same twitter trace from the experiments, and whose synthetic performance function is given by  $p_j(a, \ell) = 1/(1 + e^{-(a/\ell - b_j)})$ , where  $a$  is the allocation and  $\ell$  is the load. For the 5 users, we set  $b_j \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ . We set the SLO to be 0.95 for all users (note that  $0 \leq p_j \leq 1$ ). As the stochastic observation, we sample a Gaussian with standard deviation 0.2.



# Karma: Resource Allocation for Dynamic Demands

Midhul Vuppalapati  
Cornell University

Giannis Fikioris  
Cornell University

Rachit Agarwal  
Cornell University

Asaf Cidon  
Columbia University

Anurag Khandelwal  
Yale University

Éva Tardos  
Cornell University

## Abstract

We consider the problem of fair resource allocation in a system where user demands are dynamic, that is, where user demands vary over time. Our key observation is that the classical max-min fairness algorithm for resource allocation provides many desirable properties (*e.g.*, Pareto efficiency, strategy-proofness, and fairness), but only under the strong assumption of user demands being static over time. For the realistic case of dynamic user demands, the max-min fairness algorithm loses one or more of these properties.

We present Karma, a new resource allocation mechanism for dynamic user demands. The key technical contribution in Karma is a credit-based resource allocation algorithm: in each quantum, users donate their unused resources and are assigned credits when other users borrow these resources; Karma carefully orchestrates the exchange of credits across users (based on their instantaneous demands, donated resources and borrowed resources), and performs prioritized resource allocation based on users' credits. We theoretically establish Karma guarantees related to Pareto efficiency, strategy-proofness, and fairness for dynamic user demands. Empirical evaluations over production workloads show that these properties translate well into practice: Karma is able to reduce disparity in performance across users to a bare minimum while maintaining Pareto-optimal system-wide performance.

## 1 Introduction

Resource allocation is a fundamental problem in computer systems, spanning private and public clouds, computer networks, hypervisors, etc. There is a large and active body of research on designing resource allocation mechanisms that achieve Pareto efficiency (high resource utilization) and strategy-proofness (selfish users should not be able to benefit by lying about their demands) while ensuring that resources are allocated fairly among users, *e.g.*, [30,32,39,57,59,66,67].

For a system containing a single resource, the two most popular allocation mechanisms are strict partitioning [9,72] and max-min fairness [30,32,36,40,49,50,57,59,66].

The former allocates the resource equally across all users (“fair share”), independent of their demands; this guarantees strategy-proofness and fairness, but not Pareto efficiency since resources can be underutilized when one or more users have demands lower than the fair share. Max-min fairness alleviates limitations of strict partitioning by taking user demands into account: it maximizes the minimum allocation across users while ensuring that each user’s allocation is no more than their demand. A classical result shows that resource allocation based on max-min fairness guarantees each of the three desirable properties—Pareto efficiency, strategy-proofness, and fairness. These powerful properties have, over decades, motivated efforts in both systems and theory communities on generalizations of max-min fairness for allocating multiple resources [30–32], for incorporating application performance goals and deadlines [31,39,46,47], and for new models of resource allocation [17,22,25,33,59,66], to name a few.

This paper explores a complementary problem—resource allocation of a single elastic resource in a system where user demands are dynamic, that is, vary over time. Dynamic user demands are the norm in most real-world deployments [12,16,41,45,60,63,70,72,79]; for instance, analysis of production workloads in §2 reveals that user demands vary by as much as  $17\times$  within minutes, with majority of users having demands with standard deviation  $0.5 - 43\times$  of the average over time. We show in §2 that, for systems with such dynamic user demands, resource allocation based on the max-min fairness algorithm fails to guarantee one or more of its properties: (1) if the allocation is done based on demands at  $t=0$ , Pareto efficiency and strategy-proofness are no longer guaranteed; and, (2) if the allocation is done periodically, *long-term* fairness is no longer guaranteed—for  $n$  users with the same average demand, the max-min fairness algorithm may allocate some user as much as  $\Omega(n)$  more resources than other users over time.

We present Karma, a new resource allocation mechanism for dynamic user demands. The key technical contribution of Karma is a credit-based resource allocation algorithm: in each quantum, users receive credits when they donate a part of their fair share of resources (*e.g.*, if their demand



is less than their fair share); users can use these credits to borrow resources in any future quantum when their demand is higher than their fair share. When the supply of resources from donors is equal to the demand from borrowers, it is easy to exchange resources and credits among users. The key algorithmic challenge that Karma resolves is when supply is not equal to demand—in such scenarios, Karma carefully orchestrates resources and credits between donors and borrowers: donors are prioritized so as to keep credits across users as balanced as possible, and borrowers are prioritized so as to keep the resource allocation as fair as possible.

We theoretically establish Karma guarantees for dynamic user demands. Karma guarantees Pareto efficiency at all times: in each quantum, it allocates resources such that it is not possible to increase the allocation of a user without decreasing the allocation of at least another user. For strategy-proofness, Karma guarantees that a selfish user cannot increase their aggregate resource allocation by *over*-reporting their demands in any quantum. In addition, we show a new surprising phenomenon (that may primarily be of theoretical interest): if a user had perfect knowledge about the future demands of all other users, the user can increase its own aggregate allocation by a small constant factor by *under*-reporting its demand in some quanta; however, for  $n$  users, imprecision in this future knowledge could lead to the user losing  $\Omega(n)$  factor of their aggregate resource allocation by under-reporting their demand in any quantum. Put together, these results enable Karma to provide powerful guarantees related to strategy-proofness. Finally, for fairness, we prove that given a set of (past) allocations, Karma guarantees an optimally-fair resource allocation. We also establish that Karma guarantees similar properties even when multiple selfish users can collude, and even when different users have different fair shares.

We have realized Karma on top of Jiffy [41], an open-sourced multi-tenant elastic memory system; an end-to-end implementation of Karma is available at <https://github.com/resource-disaggregation/karma>. Evaluation of Karma over production workloads demonstrates that Karma’s theoretical guarantees translate well into practice: it matches the max-min fairness algorithm in terms of resource utilization, while significantly improving the long-term fairness of resources allocated across users. Karma’s fairer resource allocation directly translates to application-level performance; for instance, over evaluated workloads, Karma keeps the *average* performance (across users) the same as the max-min fairness algorithm, while reducing performance *disparity* across users by as much as  $\sim 2.4\times$ . Karma also incentivizes users to share resources: our evaluation shows that (1) Karma-conformant users achieve much more desirable allocation and performance compared to users who prefer a dedicated fair share of resources; and, (2) if users were to turn Karma-conformant, they can improve their performance by better matching their allocations with their demands over time.

## 2 Motivation

We begin by outlining our motivating use cases, followed by an in-depth discussion on the limitations of the classic max-min fairness algorithm for dynamic user demands.

**Motivating use cases.** Fair resource allocation is an important problem in private clouds where resources are shared by multiple users or teams within the same organization [12, 16, 17, 30–33, 36, 39, 40, 45, 46, 59, 60, 66, 70, 72, 79, 80]; our primary use cases are from such private clouds. Karma may also be useful for emerging use cases from multi-tenant public clouds where spare resources may be allocated to tenants while providing performance isolation [8, 14, 38, 41, 57, 63, 64, 66]. We discuss motivating scenarios in both contexts below.

One scenario is shared analytics clusters. For instance, companies like Microsoft, Google, and Alibaba employ schedulers [32, 35, 39, 69, 70, 80] that allocate resources across multiple internal teams that run long-running jobs (*e.g.*, for data analytics [23, 81]) on a shared set of resources. Consider memory as a shared resource; in many of these frameworks, main memory is used to cache frequently accessed data from slower persistent storage and to store intermediate data generated during job execution. Indeed, increasing the allocated memory improves job performance; however, since memory is limited and is shared across multiple teams, ensuring resource allocation fairness is also a key requirement. Moreover, since these jobs are usually long-running, their performance depends on long-term memory allocations, rather than instantaneous allocations [16, 32, 45].

Another use case is shared caches: many companies (*e.g.* Facebook [9, 12, 52] and Twitter [79]) operate clusters of in-memory key-value caches, such as memcached or Redis, serving a wide array of internal applications. In this use case, the memory demand of each application may be computed as the amount of memory that would be required to fit hot objects within the cache [18, 19, 52, 79]. In such settings, efficient and fair sharing of caches is of utmost importance [9, 19, 52, 72]: to maintain service level agreements, it is important to have consistently good performance over long periods of time, rather than excellent performance at some times and very poor performance at other times (see [9, 19, 52, 72] for more discussion on the importance of long-term performance).

Third, fair resource allocation while ensuring high utilization is also a goal in inter-datacenter bandwidth allocation [36, 40, 49]. Existing traffic engineering solutions used in production environments perform periodic max-min fair resource allocation to account for dynamic user demands [36, 40, 49]. Our work demonstrates that periodically performing max-min fair resource allocation over such dynamic demands leads to unfair resource allocation across users.

Finally, an interesting use case in the public cloud context is that of burstable VMs [2, 4] that use virtual currency to enable resource allocation over dynamic user demands. These VMs share resources with VMs from other users and are charged

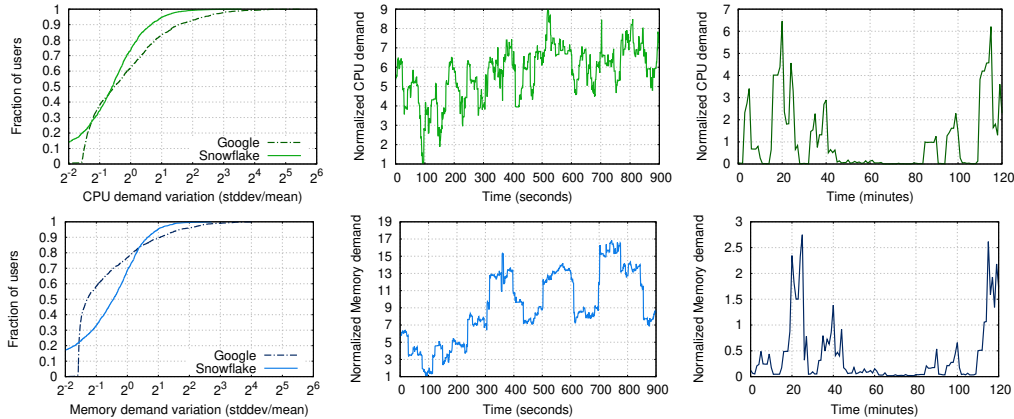


Figure 1: **Analysis of Google and Snowflake workloads suggests that a large fraction of users have dynamic demands (left) that can change dramatically over short timescales (center, right)** (Left) CDFs, across users, of the ratio of standard deviation and mean of each user’s demand. (Center) For a randomly sampled user in the Snowflake trace, the variation in the user’s CPU and memory demands (normalized by minimum demand) over a 15 minute period. (Right) For a randomly sampled user in the Google trace, the variation in the user’s CPU and memory demands (normalized by minimum demand) over a 2 hour period.

on an instance-specific baseline. When resource utilization is below the baseline, users accumulate virtual currency that they can later use to gain resources beyond the baseline during periods of high demand. Given that Burstable VMs are primarily useful for dynamic user demands, they will likely need resource allocation mechanisms that guarantee high utilization, strategy-proofness, and fair resource allocation.

**Dynamic user demands.** Increasingly many applications running data analytics or key-value caches operate on data collected from social media, application and network logs, mobile systems, etc. A unique characteristic of these data is that they are less controllable by the organization because they are generated by entities outside of the organization. As a result, applications can observe highly time-varying dynamic resource demands [12, 16, 41, 45, 60, 63, 70, 72, 79].

To build a deeper understanding of variation in user demands over time, we analyze two publicly-available production workloads: (1) Google [60] resource usage information across 8 clusters (1000–2000 users per cluster) over a 30 day period; and, (2) Snowflake [72], a cloud-based database query engine that provides resource usage statistics for over 2000 users over a 14 day period. To characterize user demand variability over time, we compute—for each user—the ratio of the standard deviation and mean of their demands over the entire period. Figure 1 (left) shows that 40–70% of all users in both Google and Snowflake workloads have a standard deviation in CPU and memory demands at least  $0.5\times$  their mean, indicating high variability in demands for most users. Furthermore, the standard deviation in demands of as many as 20% of the users can be as high as their mean demand, with some users having extremely high variance in demands (standard deviations up to 12–43 $\times$  the mean). Similar observations have been made for time-varying user demands in inter-datacenter networks; for instance,

production studies [5] show that, on average, user demands vary by 35% within 5-minute intervals, with some demands varying by as much as 45% within a short period of time.

Figure 1 (center) shows the CPU and memory demands for a randomly-sampled user from the Snowflake trace over a 15 minute window (we show only one user and only 15 minute window for clarity; analyzing a sample of 100 users, we find 87% of the users to have similar demand patterns). The figure shows that user demands can change dramatically over tens of seconds, by as much as  $6\times$  and  $2\times$  for compute and memory, respectively. Similarly, we see significant variation in demands even for a random user from the Google trace (shown in Figure 1 (right)).

**Max-min fairness guarantees fail for dynamic user demands.** The classical max-min fairness algorithm for resource allocation provides many desirable properties, *e.g.*, Pareto efficiency, strategy-proofness, and fairness. However, buried under the proofs is the assumption that user demands are static over time, an assumption that does not hold in practice (as demonstrated in Figure 1). For the realistic case of dynamic user demands, max-min fairness can be applied in two ways, each of which leads to violating one or more of its properties. We will demonstrate this using the example in Figure 2; here, time is divided into five quanta and three users have demands varying across quanta.

First, one can naïvely perform max-min fair allocation just once based on user demands at quantum  $t = 0$ . This results in max-min fairness losing both Pareto efficiency and strategy-proofness. In the example of Figure 2, since allocations will only be done based on the demands specified by the users at  $t = 0$ , if users were to specify their true demands, user C will obtain an allocation of 1 unit leading to a total useful allocation of 3 units over the entire duration (as shown in Figure 2 (middle, top)); if user C were to lie

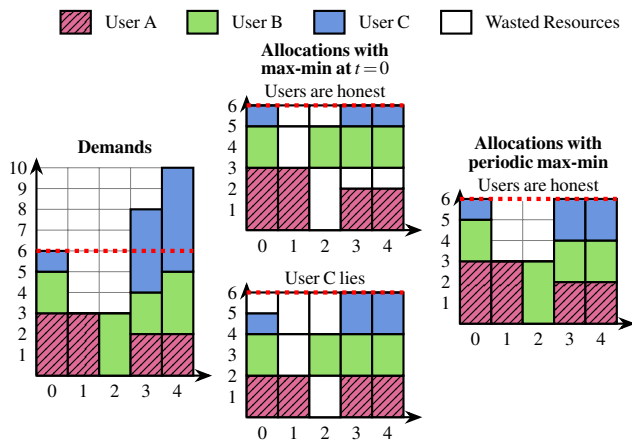


Figure 2: **Classical max-min fairness guarantees break for dynamic user demands.** Here, 6 units of a resource are shared by 3 users (fair share of 2). Discussion in §2.

and over-report their demand at  $t = 0$  as 2 units, then they can achieve a more desirable total useful allocation of 5 units (Figure 2 (middle, bottom)). This breaks strategy-proofness. In addition, max-min fairness is also not Pareto efficient: for many quanta, resources allocated to users will be underutilized as is evident in Figure 2 (middle).

A better way to apply max-min fairness for dynamic user demands is to periodically reallocate resources based on users’ instantaneous demands (e.g., every quantum of time periods, as in several operating systems and hypervisors [3, 73]). This trivially guarantees Pareto efficiency and strategy-proofness but results in extremely unfair allocation across users. Figure 2 (right, top) shows an example where max-min fairness can result in  $2\times$  disparity between resources allocated to users over the 5 quanta—user A receives a total allocation of 10 slices, while user C receives a total allocation of only 5 slices, despite them having the same average demand; this example can be easily extended to demonstrate that max-min fairness can, for  $n$  users, result in resource allocations where some user gets a factor of  $\Omega(n)$  larger amount of resources than other users (proof in [71]). Such disparity in resource allocations also leads to disparity in application-level performance across users since, as discussed above in use cases, many applications require consistently good performance over long periods of time, rather than excellent performance at some times and very poor performance at other times [22, 28, 32, 68]. We will demonstrate, in the evaluation section, that users experience significant disparity in application-level performance due to such disparate resource allocations.

For the rest of the paper, we focus on long-term fairness; informally, an allocation is considered fair if all users have the same aggregate resource allocation over time. Our goal is to design a resource allocation mechanism that, for dynamic user demands, guarantees Pareto efficiency, strategy-proofness, and fairness.

### 3 Karma

Karma is a resource allocation mechanism for dynamic user demands. Karma uses *credits* (§3.1, §3.2)—users receive credits when they donate a part of their fair share of resources (e.g., when their demand is less than their fair share), and can use these credits to borrow resources beyond their fair share during periods of high demand. Karma carefully orchestrates the exchange of resources and credits between donors and borrowers: donors are prioritized in a manner that ensures credit distribution across users remains as balanced as possible, and borrowers are prioritized in a manner that keeps the resource allocation as fair as possible. We will prove theoretically in §3.3 that, while simple in hindsight, this allocation mechanism simultaneously achieves Pareto efficiency, strategy-proofness, and fairness for dynamic user demands.

#### 3.1 Preliminaries

We consider the following setup for the problem: we have  $n$  users sharing a single resource (CPU, memory, GPUs, etc.); each user has a fair share of  $f$  resource units (each unit is referred to as a *slice*), and thus the pool has  $n \times f$  slices of the resource (as we discuss in §3.4, all our results hold for users having different fair shares). Time is divided into quanta, users demand a certain number of resource slices every quantum, and Karma performs resource (re)allocation at the beginning of each quantum. While user demands during each quantum can be arbitrary, unsatisfied demands in one quantum do not carry over to the next. Similar to prior work [30, 57, 59, 66], we assume that users are not adversarial (that is, do not lie about their demands simply to hurt others’ allocations), but are otherwise selfish and strategic (willing to misreport their demands to maximize their allocations).

#### 3.2 Karma design

Let  $0 \leq \alpha \leq 1$  be a parameter. Karma guarantees that each user is allocated an  $\alpha$  fraction of its fair share ( $= \alpha \cdot f$ ) in each quantum; we refer to this as the guaranteed share. Karma maintains a pool of resource slices—karmaPool—that, at any point in time, contains two types of slices:

- **Shared slices** are the slices in the resource pool that are not guaranteed to any user. It is easy to see that the number of shared slices in the system is  $n \cdot f - n \cdot \alpha \cdot f = n \cdot (1 - \alpha) \cdot f$ .
- **Donated slices**, that are donated by users whose demands are smaller than their guaranteed share.

We use these two sets of slices in the following manner. In any given quantum, if a user has demand less than its guaranteed share, then the user is said to be “donating” as many slices as the difference between the user’s guaranteed share and demand in that quantum. A user that has demand larger than its guaranteed share is said to be “borrowing” slices beyond its guaranteed share, which the system can potentially supply using either shared slices or donated slices.

### 3.2.1 Karma credits

Karma allocates resources not just based on users' instantaneous demands, but also based on their past allocations. To maintain past user allocation information, Karma uses credits.

Users earn credits in three ways. First, each user is bootstrapped with a fixed number of initial credits upon joining the system (we discuss the precise number once we have enough context, in §3.4); second, each user is allocated  $(1 - \alpha) \cdot f$  free credits every quantum as compensation for contributing  $(1 - \alpha)$  fraction of its fair share to shared slices. Finally, users earn one credit when some other user borrows one of their *donated* slices (one credit per quantum per slice).

Unlike earning credits, there is only one way for any user to lose credits: for every slice borrowed from the karmaPool (donated or shared), the user loses one credit.

### 3.2.2 Prioritized resource allocation

We now describe Karma's resource allocation algorithm, that orchestrates resources and credits across users (Algorithm 1). To make the discussion succinct, we refer to the sum of user demands beyond their guaranteed share as "borrower demand"; that is, to compute borrower demand for any given quantum, we take all users with demand greater than their guaranteed share and sum up the difference between their demand (in that quantum) and  $\alpha \cdot f$ . In quanta when borrower demand is equal to the supply (number of slices in karmaPool), Karma's decision-making is trivial: simply allocate all slices in karmaPool to the borrowers, and update credits for all users as described in the previous subsection. The key algorithmic challenge that Karma resolves is when the supply is either more or less than the borrower demand. We describe Karma allocation mechanism for such scenarios next and then provide an illustrative example.

**Orchestrating resources and credits when supply > borrower demand.** When supply is greater than borrower demand, there are enough slices in karmaPool to satisfy the demands of all borrowers. In such a case, Karma prioritizes the allocation of donated slices over shared slices (so that donors get credits), and across multiple donated slices, prioritizes the allocation of a slice from the donor that has the smallest number of credits—this allows "poorer" donors to earn more credits, and moves the system towards a more balanced distribution of credits across users. Intuitively, credits capture the allocation obtained by a user until the last quantum—users who obtained lower allocations in the past will have a higher than average (across users) number of credits, while those who received a surplus of allocations will have a below-average number of credits. Hence, balancing the number of credits across users over time allows Karma to move towards a more equitable set of total allocations across users. Once all donated slices are allocated, Karma allocates shared slices to satisfy the remaining borrower demands.

---

#### Algorithm 1: Karma resource allocation algorithm.

---

`demand[u]`: demand of user `u` in the current quantum  
`credits[u]`: credits of user `u` in the current quantum  
`alloc[u]`: allocation of user `u` in the current quantum  
`f`: fair share  
 `$\alpha$` : guaranteed fraction of fair share

Every quantum do:

```
1: shared_slices  $\leftarrow n \cdot (1 - \alpha) \cdot f$ 
2: For each user u,
3:   increment credits[u] by  $(1 - \alpha) \cdot f$ 
4:   donated_slices[u] =  $\max(0, \alpha \cdot f - \text{demand}[u])$ 
5:   alloc[u] =  $\min(\text{demand}[u], \alpha \cdot f)$ 
6: donors  $\leftarrow$  all users u with donated_slices[u] > 0
7: borrowers  $\leftarrow$  all users u with
8:   alloc[u] < demand[u] & credits[u] > 0
9: while borrowers  $\neq \emptyset$  and
10:    $(\sum_u \text{donated\_slices}[u] > 0 \text{ or } \text{shared\_slices} > 0)$ 
11: do
12:   b*  $\leftarrow$  borrower with maximum credits
13:   if donors  $\neq \emptyset$  then
14:     d*  $\leftarrow$  donor with minimum credits
15:     Increment credits[d*] by 1
16:     Decrement donated_slices[u] by 1
17:     Update the set of donors (line 6)
18:   else
19:     Decrement shared_slices by 1
20:   Increment alloc[b*] by 1
21:   Decrement credits[b*] by 1
22:   Update the set of borrowers (line 7)
```

---

**Orchestrating resources and credits when supply < borrower demand.** When supply is less than demand, karmaPool does not have enough slices to satisfy all borrower demands. In such a scenario, Karma prioritizes allocating slices to users with the maximum number of credits. This strategy essentially favors users that had fewer allocations in the past (and thus, a larger number of credits), hence moving the system towards a more balanced allocation of resources across users, promoting fairness. At the same time, reducing the credits for the users with the most credits also moves the system to a more balanced distribution of credits across users.

**Illustrative example.** We now illustrate through a concrete example. The running example in Figure 3 shows the execution of Karma's algorithm for the example from Figure 2 for  $\alpha = 0.5$ : that is three users A, B, and C, each with a fair share 2 slices ( $f = 2$ ), and a guaranteed share of 1 slice. Recall that, since  $(1 - \alpha) \cdot f = 1$ , each user receives 1 credit every quantum, and suppose all users are bootstrapped with 6 initial credits.

In the first quantum, C's demand is equal to the guaranteed share, while A and B request 2 and 1 slices beyond the guaranteed share, respectively. Since supply (= 3 shared slices in karmaPool) is equal to borrower demand, Karma uses the shared slices to allocate slices beyond the guaranteed share for



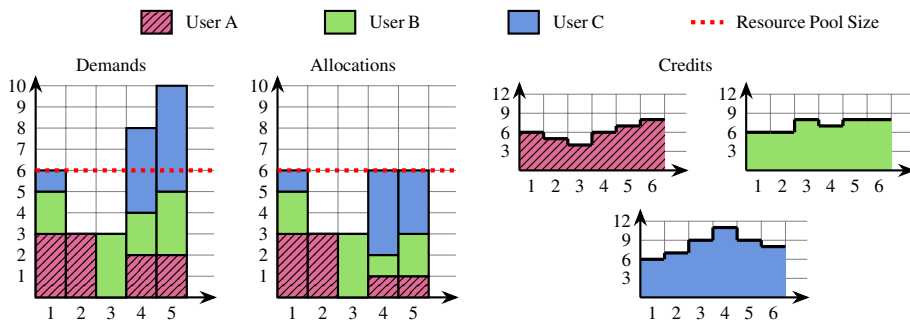


Figure 3: **Karma resource allocation for the running example of Figure 2:** Recall that there are 6 resource slices, 3 users each with average demand and fair share equal to 2. We show the case of the guaranteed share being 1 ( $\alpha=0.5$ ), with 6 bootstrapping (initial) credits for each user. Note that each user receives 1 free credit every quantum. Karma achieves significantly improved fair allocation than max-min fairness—it allocates each user an equal allocation of 8 resource slices over time.

A and B and satisfies their demands. This results in a final allocation of 3 slices for A, 2 slices for B, and 1 slice for C. A loses 2 credits, and B loses 1 credits, and no one gains any credits.

In the second quantum, A demands 3 slices, while B and C donate 1 slice each. The total supply ( $=5$ , with 2 donated slices and 3 shared slices) exceeds the borrower demand. A is allocated 3 slices and it loses 3 credits (since its allocation is 2 slices above its guaranteed share). B and C receive 1 credit each since their donated slices are used. Similarly, in the third quantum, B demands 3 slices, while A and C donate 1 slice each. Since total supply exceeds borrower demand, B receives the 3 slices it asked for, and loses 2 credits; A and C gain 1 credit each.

The fourth quantum is important: here, demand exceeds supply, and there are no donated slices. Now, unlike classic max-min fairness, Karma will prioritize the allocation of resources based on the credits of each tenant. Since at the start of this quantum, C has 11 credits, while A and B have only 6 and 7 credits respectively, C will be able to get 3 extra slices from the pool of shared slices by using 3 credits and achieve an allocation of 4. A and B will get their guaranteed allocation of 1 and do not gain or lose any credits.

In the fifth quantum, once again, demand exceeds supply. C has 9 credits, B has 8 credits, and A has 7 credits. Karma first prioritizes allocating to C giving it 1 extra slice, at which point both C and B have equal credits (8). Next, they both get 1 extra slice each, at which point the supply is exhausted. The final resulting allocation is 1 slice for A, 2 slices for B, and 3 slices for C.

In the end, A, B, and C end up with the exact same total allocation (8 slices) and number of credits (unlike max-min fairness where user allocations had a disparity of  $2\times$ ).

### 3.3 Karma Properties & Guarantees

In this section, we present a theoretical analysis of Karma. Recall from §3.1 that, similar to all prior works, users are considered selfish and strategic (that is, are willing to misreport their demands to maximize their allocations), but not adversarial (that is, do not lie about their demands simply

to hurt others' allocations). For the purpose of our theoretical analysis, we assume that Karma is initialized with a large enough number of initial credits so that users do not run out of credits during the execution of the algorithm (we discuss how to achieve this in practice in § 3.4). All our results hold for  $\alpha = 0$ ; extending our results to  $\alpha > 0$  is an interesting open question. Finally, while we provide inline intuition for each of our results, full proofs are presented in [71].

We define Pareto efficiency on a per-quantum basis. An allocation is said to be Pareto efficient if it is not possible to increase the allocation of a user without decreasing the allocation of at least one other user by a similar total amount during that quantum. Note that, Pareto efficiency on a per-quantum basis implies Pareto efficiency over time.

**Theorem 1.** *Karma is Pareto efficient.*

Karma's Pareto efficiency follows trivially from the observation that similar to max-min fairness, Karma allocation satisfies the two properties: (1) no user is allocated more resources than its demand, and (2) either all resources are allocated or all demands are satisfied.

For strategy-proofness, we make two important notes. First, if one assumes that the system has a priori knowledge of all future user demands, the resource allocation problem can be solved trivially using dynamic programming; however, for many use cases, it is hard to have a priori knowledge of all future user demands. This leads to our second note: Karma is solving an “online” problem (that is, it does not assume a priori knowledge of future user demands), and thus, we prove online strategy-proofness [7] defined as follows: assume that all users are honest during quanta 0 to  $q - 1$ ; then, a mechanism is said to be online strategy-proof if, for any quantum  $q$ , a user cannot increase its allocation during quantum  $q$  by lying about its demand during quantum  $q$ .

**Theorem 2.** *Karma is online strategy-proof.*

To prove Theorem 2, we actually prove a stronger result stated below. Karma's online strategy-proofness trivially follows from this.

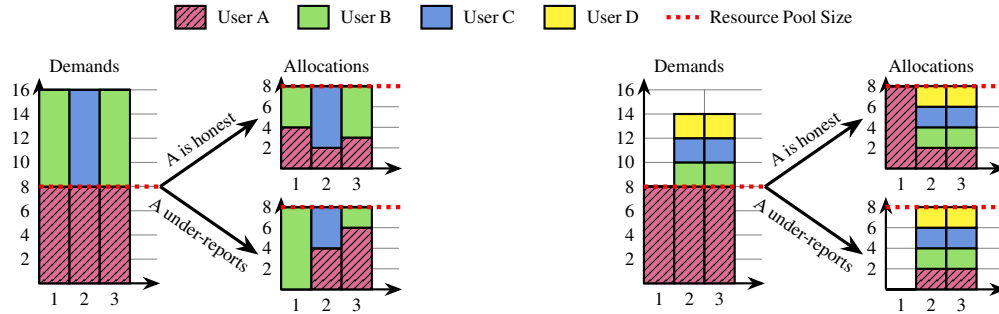


Figure 4: The phenomenon of users (left) gaining a small factor of improvement in their allocations by specifying demands less than their real demands, by exploiting knowledge of all future demands of all users; (right) any imprecision in the knowledge of future demands of all users could result in a significant reduction in useful allocations of the lying user. The resource pool has 8 slices, and 4 users with fair share of 2 and guaranteed share of 0 ( $\alpha=0$ ).

**Lemma 1.** *A user cannot increase its useful resource allocation by specifying a demand higher than its real demand in any quantum.*

The proof for the lemma is a bit involved, but intuitively, it shows the following. The immediate effect of a user specifying a demand higher than its actual demand is that if the user is allocated more resources than its actual demand, these extra resources do not contribute to its utility, but do put the user into a disadvantageous position: not only can this user lose credits (either because it's asking for resources beyond its guaranteed share, or because it could have gained credits if this extra resource could have been allocated to some borrower), but also because other users get fewer resources; this makes other users be favored by the allocation algorithm in the future while making the lying user less favored. Thus, the user cannot increase its long-term “useful” allocation by specifying a demand higher than the real demand in any quantum. Specifically, it is possible that when a user over-reports its demand during quantum  $q'$ , the user receives an increased instantaneous allocation during some future quantum  $q > q'$ ; however, we are able to show that, in this case, the user will also receive reduced instantaneous allocation(s) during other quantum(s) in between  $q'$  and  $q$ , leading to either a lower or equal total allocation over the period between  $q'$  and  $q$ . The hardness in the proof stems from carefully analyzing such cascade effects: a small change in users' resource allocation in any quantum can result in complex changes in future allocations that may lead to higher instantaneous but equal or lower total allocations in future quanta. Once we prove this lemma, the proof for Karma's online strategy-proofness follows immediately.

While analyzing Karma properties, we encountered a new, surprising, phenomenon that may be of further theoretical interest: we show that a user that *knows all future demands of all other users* can report a demand that is lower than its actual demand in the current quantum to increase its allocation in future quanta by a small constant factor. However, any imprecision in the knowledge of all future demands of all other users could result in the user losing a factor of  $\Omega(n)$  of its total allocation.

**Lemma 2.** *A user cannot increase its total useful allocation by a factor more than  $1.5 \times$  by specifying a demand less than its real demand in any quantum. Gaining this useful allocation requires the user to know the future demands of all users. If the user does not have a precise knowledge of all future demands of all users, it can lose its useful allocation by a factor of  $\frac{n+2}{2}$  (for  $n \geq 3$ ) by specifying a demand less than its real demand.*

We provide intuition for this phenomenon using an example (Figure 4). In the left figure, user A is able to gain 1 extra slice in its overall allocation by under-reporting its demand (reporting 0 instead of 8) in the first quantum. By under-reporting, its allocation in the first quantum reduces, enabling it to get more resources during the second quantum when it competes with user C. In the third quantum, it is able to recover the resources it lost in the first quantum from user B, resulting in an overall gain. To see the flip-side, if the demands of other users had been as shown in Figure 4 (right), then user A sees a  $3 \times$  degradation in overall allocation.

To prove the first part of the Lemma 2, we consider an arbitrary user Alice and an arbitrary time period, and compare two scenarios—one where Alice is truthful (hereby called the truthful scenario) and one where Alice is deviating by under-reporting her demand during some quantum (hereby called the deviating scenario).

Our key insight for the proof is that bounding the increase in total allocation of *all users* is easier than reasoning about the increase in total allocation of an individual user (Alice) since even a small change in Alice's demand during one quantum can result in cascading effects on the total allocation of other users as well. To that end, we prove the following claim: the total amount of resources all the users have earned in excess in the deviating scenario compared to the truthful one can be at most as large as Alice's total allocation in the truthful scenario. We prove this claim based on the following observation: whenever Alice under-reports her demand she is effectively “donating” the allocation she would have gotten in the truthful scenario to the other users whose allocations in the deviating scenario increase. Since Karma is Pareto efficient, the total

gain in allocation across users during this quantum is limited by the amount donated by Alice which is in turn bound by Alice’s own allocation during this quantum in the truthful scenario. By applying this reasoning iteratively across all quanta<sup>1</sup>, we can show that the total increase in allocation across all users cannot exceed the total allocation of Alice in the truthful scenario. This already implies a  $2\times$  upper bound on the maximum increase in total allocation that Alice can achieve.

To tighten the upper bound, we prove a second claim: if Alice receives higher total allocation in the deviating scenario compared to the truthful scenario, then there must exist some other user Bob who gained an even larger increase in total allocation than Alice. Putting together the above two claims allows us to establish the desired upper bound. Based on the first claim, the total gain in allocation across all users cannot exceed Alice’s total allocation in the truthful scenario. This implies that the sum of total gains across Alice and Bob cannot exceed Alice’s total allocation in the truthful scenario. Since Bob’s gain is at least as large as Alice’s gain (based on the second claim), this implies that Alice’s gain is at most half the total allocation of Alice in the truthful scenario—a gain of at most  $1.5\times$ , thus proving the first part of Lemma 2.

The second part of the lemma is proven by first creating a set of demands where a user can under-report its demand during quantum  $q$  to earn increased total allocation by some quantum  $q' > q$ . Then we create a set of demands that are identical up to quantum  $q$  but vastly different from quanta  $q+1$  to  $q'$ . If the user (in the hope of facing the first set of demands) under-reports its demand on quantum  $q$  but ends up facing the second set of demands then this results in vastly different allocations by quantum  $q'$ . By correctly picking the two sets of demands we get the desired bounds.

In [71], we prove an even stronger result that extends Karma properties from Theorem 1, Theorem 2, Lemma 1 and Lemma 2 to the case of multiple colluding users:

**Theorem 3.** *No group of colluding users can increase their allocation by specifying a demand higher than their real demand. Additionally, for any group of colluding users, under-reporting demands cannot lead to more than a  $2\times$  improvement in their useful resource allocation. Finally, even if users form coalitions, Karma is Pareto efficient and online strategy-proof.*

Recall that Karma focuses on long-term fairness without a priori knowledge of future user demands. To that end, the following theorem summarizes Karma’s fairness guarantees:

**Theorem 4.** *For any quantum  $q$ , given fixed user allocations from quantum 0 to quantum  $q - 1$ , and user demands at quantum  $q$ , Karma maximizes the minimum total allocation from quantum 0 to quantum  $q$  across users.*

<sup>1</sup>It turns out that Alice under-reporting in a given quantum cannot cause cascading increases in total allocation across users in future quanta if Alice does not under-report in future quanta. This is because Karma prioritizes allocation to users with high credits (or equivalently low total allocations).

The proof for the above theorem follows from the prioritized resource allocation mechanism of Karma. Intuitively, given allocations from quantum 0 to  $q - 1$ , the user with the least total allocation up to quantum  $q - 1$  will have the largest number of credits. In quantum  $q$ , Karma will prioritize the allocation of resources to this user (until it is no longer the one with the minimum total allocation, after which it will prioritize the next user with the minimum total allocation, and so on), thus maximizing the minimum total allocation from quantum 0 to  $q$  across users—this is the best one can do in quantum  $q$  given past allocations.

### 3.4 Discussion

Finally, we briefly discuss some additional aspects of Karma design not included in the previous subsections.

**Bootstrapping Karma with initial credits.** Recall that, to bootstrap users, Karma allocates each user an initial number of credits. The precise number of initial credits has little impact on Karma’s behavior; after all, credits in Karma essentially capture a relative ordering between users, rather than having any absolute meaning. The only importance of the number of credits is to ensure that no user runs out of credits at any quantum (which, in turn, could lead to violation of Karma’s Pareto efficiency guarantees): even if spare resources are available, a user with high demand may not be able to borrow resources beyond the guaranteed share (line 7 of Algorithm 1) due to running out of credits. Thus, Karma sets the number of initial credits to a large numerical value to ensure that no user ever runs out of credits<sup>2</sup>,

**User churn.** Fairness is relatively ill-defined when users can join and leave the system on a short-term basis (*e.g.*, when a user runs a short query with large parallelism, and then leaves the cluster). Also, recall from our motivating scenarios, fair resource allocation in private clouds is usually performed for long-running services. However, Karma still handles user churn since, in many realistic scenarios, the set of all users of the system may not be known upfront during system initialization. For users that join and leave over longer timescales, Karma handles user churn with a simple mechanism: its credits. When a new user joins, either the resource pool size remains fixed and the fair share of all users is reduced proportionally or the resource pool size increases and the fair share of users remains the same. The credits of the existing  $n - 1$  users do not change, and the new user is bootstrapped with initial credits equal to the current average number of credits across the existing  $n - 1$  users. Intuitively, users who have donated more resources than they have borrowed will have above-average credits, and those who have borrowed

<sup>2</sup>For example, in a system with 100 users with fair share of 100 slices, setting initial credits to say  $10^{13}$  will ensure that even a worst-case user with highest possible demand (10000 slices) during all quanta cannot run out of credits for  $\sim 31$  years, which is good enough for all practical purposes.

more than they have donated will have below-average credits. As such, initializing the new user with the average number of credits (heuristically) puts the new user on equal footing with an existing user that has borrowed and donated equal amounts of resources over time. When a user leaves the system, the fair share of the remaining users is increased proportionally (or resource pool size reduces while maintaining the same fair share), and there is no change in their credits.

**Users with different fair shares.** We have presented Karma’s algorithm for the case of users having the same fair share merely for simplicity: all our results extend to the case of users having different fair shares. To generalize the algorithm to users with different fair shares, users with larger weights are charged fewer credits to borrow resources beyond their guaranteed share when compared to users with smaller weights. Intuitively, this enables users with larger weights to obtain more resources than users with smaller weights for the same number of credits. We achieve this by updating Line 20 of Algorithm 1 to decrement credits by  $\frac{1}{n \cdot w_i}$  instead of 1, where  $w_i$  is the normalized weight of the corresponding user, and  $n$  is the number of users. For users with different fair shares, this generalization leads to the same properties and guarantees as discussed in §3.3 (the only difference, is that the upper bound factor in Lemma 2 changes from  $1.5\times$  to  $2\times$ ). A full description of the weighted version of the algorithm along with proofs of guarantees can be found in [71].

**System parameters, and interpretation for  $\alpha$ .** Karma has only one parameter:  $\alpha$ ; one can think of resource slice size and quantum duration as parameters, but these are irrelevant to Karma’s guarantees: they hold for any slice size and quantum duration, as long as demands change at coarse timescales than the quantum duration. The  $\alpha$  parameter in Karma provides a tradeoff between instantaneous and long-term fairness. Providers can choose any  $\alpha$  depending on the desired properties. Intuitively, an  $\alpha$  smaller than 1 leads to a larger portion of shared slices, giving Karma’s algorithm more flexibility in adjusting allocations to achieve better long-term fairness.

## 4 Karma Implementation Details

We have implemented Karma on top of Jiffy [41], an open-sourced elastic far memory system. Jiffy has a standard distributed data store architecture (Figure 5(a)): resources are partitioned into fixed-sized slices (blocks of memory) across a number of resource servers (memory servers), identified by their unique sliceIDs (referred to as blockIDs in Jiffy). A logically centralized controller tracks the available and allocated slices across the various resource servers and stores a mapping that translates sliceIDs to the corresponding resource server. We have implemented Karma as a new resource allocation algorithm at the Jiffy controller<sup>3</sup>.

<sup>3</sup>Karma can thus directly piggyback on Jiffy’s existing mechanisms for controller fault tolerance [41, Section 4] to persist its state across failures.

Users interact with the system through a client library that provides APIs for requesting resource allocation and accessing allocated resource slices. Users express their demands to the controller through resource requests which specify the number of slices required. The controller periodically performs resource allocation using the Karma algorithm and provides users with the sliceIDs of the resource slices that are allocated to them. Users can then directly access these slices from the resource servers through read or write API calls without requiring controller interposition. In the rest of this section, we discuss the key data structures and mechanisms required to integrate Karma with Jiffy.

Karma employs three key data structures to efficiently implement the policies and mechanisms outlined in §3: karmaPool, a credit map, and a rate map.

**karmaPool.** Recall from §3.2 that the karmaPool tracks the pool of donated slices and shared slices, and needs to be updated when resource allocations change. Also, the resource allocation algorithm should be able to efficiently select donated slices from a particular user while satisfying borrower demands (§3.2.2). To this end, the karmaPool is implemented as a hash map, mapping userIDs to the list of sliceIDs corresponding to slices donated by them. The list of sliceIDs corresponding to shared slices is stored in a separate entry of the same hash map. When resource allocations change, the corresponding sliceIDs are added to or removed from the corresponding lists. As such, karmaPool supports all updates in  $O(1)$  time.

**Credit Tracking.** Karma employs two data structures for tracking and allocating credits across various users: a rate map and a credit map. The rate map maps each user to the *rate* at which it earns or spends its credits every quantum, that is, the difference between the user’s guaranteed share and the number of its allocated slices in that quantum. The rate is positive when the user is earning (that is, has donated slices) and negative when it is spending credits (that is, has borrowed slices), respectively. The credit map, on the other hand, maps each user to a counter corresponding to its current credits.

Separating the rate map and credit map facilitates efficient credit tracking at each quantum: Karma simply iterates through the rate map entries, and updates the credit counters in the credit map based on the corresponding user credit rates. Since the rate map only contains entries for users with non-zero rates, Karma can efficiently update credits for only the relevant users. At the same time, Employing a hash-map for each of them permits  $O(1)$  updates to the user credit rate or number of credits while performing resource allocation.

**Borrowing and donating slices.** Karma realizes its credit-based prioritized allocation algorithm (§3.2) using two modules at the controller. First is a *slice allocator* that maintains the karmaPool to track and update slice allocations across users, and, second a *credit tracker* that maintains the current number of credits for any user (via Credit Map) and



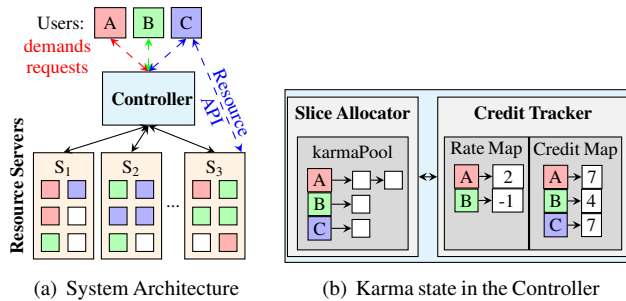


Figure 5: **Karma Design.** See §4 for details.

how it should be updated (via Rate Map). Figure 5(b) shows these modules along with the data structures they manage.

The slice allocator intercepts resource requests from users, periodically executes the Karma resource allocation algorithm (Algorithm 1) to compute allocations based on the user demands, and updates slices in the karmaPool accordingly. It interacts with the credit tracker to query and update user credits. A naïve implementation of Algorithm 1 runs in  $O(n \cdot f \cdot \log n)$  time, where  $n$  is the number of users, and  $f$  is the fair share<sup>4</sup>. Instead of computing allocations one slice at a time, we use an optimized implementation that carefully computes them in a batched fashion (full details are provided in [71]). This enables the slice allocator to support resource allocation at fine-grained timescales.

**Consistent hand-off of resources.** Since users are allowed to directly access slices from resource servers, we need to ensure consistent hand-off of slices from one user to another when slices are reallocated. For example, say user  $U_1$  has a slice during a given quantum, and in the next quantum, this slice is allocated to user  $U_2$ . We need to ensure that (1)  $U_1$ 's data is flushed to persistent storage before  $U_2$  overwrites it (2)  $U_1$  should not be able to read/write to the slice after  $U_2$  has accessed it (for example, there could be in-flight read/write requests to the slice which were initiated before  $U_1$  gets to know it's allocation changed).

Karma ensures the above by maintaining a monotonically increasing sequence number and current userID for each slice, at both the controller (within the karmaPool) and the resource servers (as slice metadata). On slice allocation, its userID is updated and its sequence number is incremented at the controller, and the sequence number is returned to the user. Subsequent user reads and writes to the slice specify this userID and sequence number. A slice read succeeds only if the accompanying sequence number is the same as the current slice sequence number, while a slice write succeeds only if the accompanying sequence number is the same or greater than the current sequence number. If a write necessitates an overwrite of the current slice content and metadata, the

<sup>4</sup>The loop in Line 10 of Algorithm 1 takes  $O(n \cdot f)$  iterations and each iteration would take  $O(\log n)$  time to find the donor/borrower with the minimum/maximum credits (if we were to maintain min/max heaps for the donor and borrower sets).

old slice content is transparently flushed persistent storage (e.g., S3) before the overwrite. In our example above,  $U_2$ 's first access to the slice after re-allocation will trigger a flush of  $U_1$ 's data to S3 and update the slice sequence number. Following this  $U_1$ 's accesses to this slice will fail since the current sequence number of the slices is higher.  $U_1$  can then read/write this data from persistent storage. Implementing consistent resource hand-off in Jiffy required minor changes to the controller (to track sequence numbers per slice), memory servers (to perform sequence number checking), and the client library (to tag requests with sequence numbers).

## 5 Evaluation

We have already established Karma properties theoretically in §3. In this section, we evaluate how Karma's properties translate to application-layer benefits over an Amazon EC2 testbed with real-world workloads. Our evaluation demonstrates that:

- Karma reduces the performance disparity between different users by  $\sim 2.4\times$  relative to classic max-min fairness, without compromising on system-wide utilization or average performance (§5.1);
- Karma incentivizes users to share resources, quantifying Karma's online strategy-proofness property (§5.2);

We primarily focus on the shared cache use case from §2 for the following reason. While datasets for the shared data analytics clusters use case are publicly available (e.g., Google and Snowflake datasets), they do not provide user queries that may impact our final conclusions. For the shared cache use case, we do have all the information we need: these datasets provide information on the working set size of each user over time, which can be fed into an end-to-end multi-tenant in-memory cache system running on Amazon EC2. We, thus, focus on this use case.

**Experimental setup.** Our experimental setup consists of a distributed elastic in-memory cache shared across multiple users backed by a remote persistent storage system. For the cache, we use Jiffy [41], augmented with our implementation of Karma (§4) and other evaluated schemes. If the evaluated scheme does not allocate sufficient slices to a user on Jiffy to fit its entire working set, the remaining data is accessed from remote persistent storage. When slices are reallocated between users across quanta, the corresponding data is moved between Jiffy and persistent storage through the consistent hand-off mechanism described in §4. We deployed our setup on Amazon EC2 using c5n.9xlarge instances (36 vCPUs, 96GB DRAM, 50Gbps network bandwidth). We host the Jiffy controller and resource servers across 7 instances and use 25 instances for the users/clients that issue queries to Jiffy. We use Amazon S3 as the persistent storage system.

**Workload.** We use the publicly available Snowflake dataset [72] that provides dynamic user demands in terms of memory usage for each customer from Snowflake's

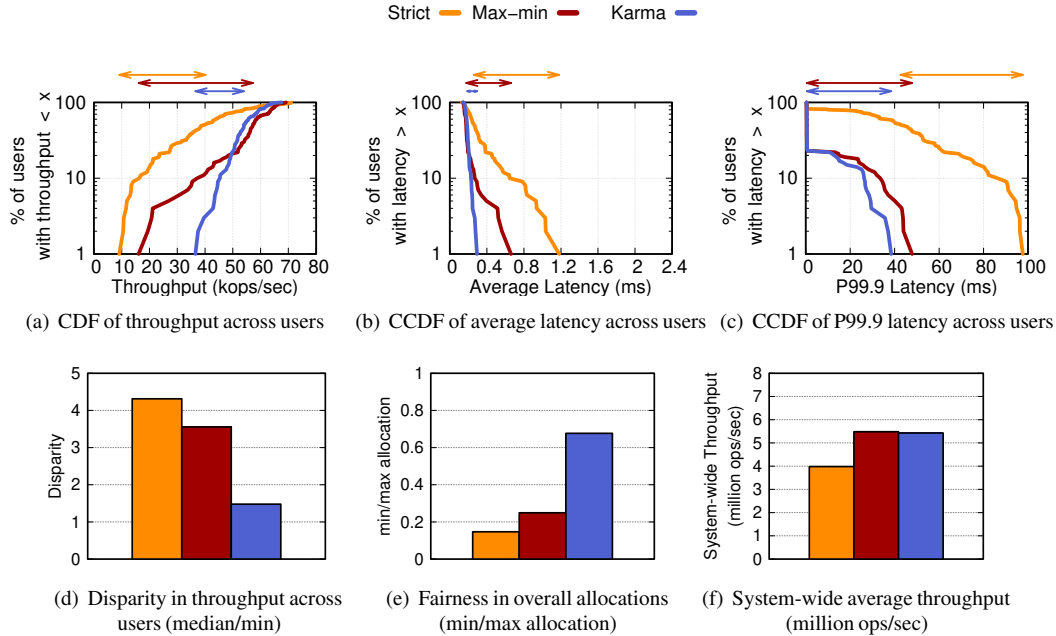


Figure 6: **Understanding Karma benefits.** (a) Karma enables a much tighter throughput distribution across users (colored arrows show the absolute gap between median and minimum throughput across users). (b, c) It also enables a tighter distribution of average and tail latencies across users (again, colored arrows show the absolute gap between median and maximum latency across users). (d) Karma achieves much lower throughput disparity—ratio of median to minimum values of throughput across users—than classic max-min fairness. (e) It also significantly reduces the gap between the users with minimum and maximum overall allocations, (f) while achieving similar system-wide performance as max-min fairness.

production cluster. We use these demands as the dynamic working set size for individual users. For each user, we issue data access queries using the standard YCSB-A workload [20] (50% read, 50% write) with uniform random access distribution, with queries during each quantum being sampled (according to the YCSB parameters) within the instantaneous working set size of that user. If a query references data that is currently cached in Jiffy, then it is serviced directly from the corresponding resource server; otherwise, it is serviced from the persistent storage.

**Default parameters.** Unless specified otherwise, we randomly choose 100 users (out of  $\sim 2000$  users) over a randomly-chosen 15 minute time window (out of a 14-day period) in the Snowflake workload. To test for extreme scenarios, we set the length of each quantum to be one second (that is, a total of 900 quanta). The fair share of each user is 10 slices, and the total memory capacity of the system is set to the number of users times the fair share (1000 slices). Each slice is 128MB in size, while each query corresponds to a read or write to a 1KB chunk of data (the default size in the YCSB workload).

**Compared schemes.** We compare Karma to strict partitioning and max-min fairness, since they correspond to the two most popular fair allocation schemes, and represent extremes in resource allocation and performance. When evaluating Karma,

we set the number of initial credits to a large value<sup>5</sup>. The fraction of fair share that is guaranteed ( $\alpha$ ) is 0.5 by default.

**Metrics.** We evaluate system-wide resource utilization, along with both per-user and system-wide performance—key metrics for any resource allocation mechanism. For performance, we measure both throughput and latency (average and 99.9th percentile tail). We define performance *disparity* for an allocation scheme as the ratio of median to minimum performance (that is, throughput or latency) observed across various users. For any given user, we define *welfare* over time  $t$  as  $\frac{\sum_t \text{allocations}}{\sum_t \text{demands}}$ , that is, the fraction of its total demands satisfied by the allocation scheme. We define *fairness* as  $\frac{\min_{\text{users}} \text{welfare}}{\max_{\text{users}} \text{welfare}}$  (higher is better, 1 is optimal), as a measure of welfare disparity between users.

## 5.1 Understanding Karma Benefits

We now evaluate Karma’s benefits in terms of reducing disparity across users’ application-level performance as well as resource allocation.

**Karma reduces performance disparity between users.** Figure 6(a) shows the throughput distribution across users for our compared schemes; the y-axis is presented in log-scale to

<sup>5</sup>As discussed in §3.4, the precise value is unimportant. Here, we set it to 900,000, so that even if a user was allocated the full system capacity for the entire duration ( $1000 \times 900$ ) it would not run out of credits.

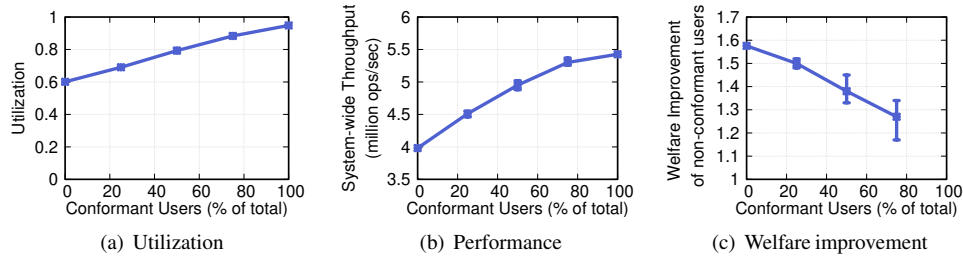


Figure 7: **Karma incentivizes resource sharing.** All metrics are computed as averages (with error bars) for three random selections of users being non-conformant. See §5.2 for details.

focus on the users at the tail of the distribution, which observe the most performance disparity. Since Karma strives to balance fairness over time, it significantly narrows the throughput distribution across users compared to the two baselines: the ratio between the maximum and minimum throughput across all users is  $7.8\times$  with strict partitioning and  $4.3\times$  with max-min fairness, but only  $1.8\times$  for Karma. As Figure 6(d) shows, Karma lowers the throughput disparity across users by  $2.4\times$  compared to max-min fairness. Karma also reduces average latency disparity (Figure 6(b)) by  $2.4\times$  and 99.9th percentile latency disparity (Figure 6(c)) by  $1.2\times$  compared to max-min fairness by enabling a tighter distribution for both latencies.

Equitability in performance across users for a scheme is closely tied to how fairly resources are allocated across users. Specifically, because of the large gap between elastic memory (Jiffy) and S3 latencies ( $50\text{--}100\times$ ), accesses to slices in S3 result in significantly lower throughput than accesses to slices in elastic memory. As a result, users’ average throughput ends up being roughly proportional to their total allocation of slices in elastic memory over time. Similarly, since a larger total allocation results in a smaller fraction of requests going to S3, average and tail latencies also reduce.

**Karma reduces disparity in allocations.** We now quantify disparities in overall allocations obtained by users across our compared schemes via our fairness metric in Figure 6(e). Due to dynamic demands, strict partitioning exhibits very poor fairness, since users with very bursty demands end up getting much lower total allocations than users who have steady demands<sup>6</sup>. While, max-min fairness observes better fairness compared to strict partitioning, the best-off user still receives  $4\times$  higher allocation than the worst-off user, resulting in poor absolute fairness. Karma achieves significantly better fairness with the best-off user receiving only  $1.5\times$  higher allocation than the worst-off user. It is able to achieve this by prioritizing the allocation of resources beyond the fair share to users with more credits (§3.2.2).

**Karma achieves Pareto efficiency and high system-wide performance.** Karma achieves the same overall resource

<sup>6</sup>Note that only *useful* allocations are considered—strict partitioning guarantees a fixed allocation at all times, but resources may remain unused when demand is low.

utilization as max-min fairness ( $\sim 95\%$ ). This is because Karma is Pareto efficient (§3.3) similar to max-min fairness and thus achieves near-optimal utilization. We find that the optimal utilization is  $< 100\%$  since some quanta observe total user demands less than system capacity.

Max-min fairness observes  $1.4\times$  higher system-wide throughput (that is, throughput aggregated across all users) than strict partitioning (Figure 6(f)) since it permits allocations beyond the fair share, allowing more requests to be served on faster elastic memory. Karma observes system-wide performance similar to max-min fairness for similar reasons; the slight variations are attributed to variance in S3 latencies.

## 5.2 Karma Incentives

We now empirically demonstrate that Karma incentivizes users to donate resources instead of hoarding them, to improve their own as well as overall system welfare. To this end, we vary the fraction of users using Karma that are *conformant* or *non-conformant*. A conformant user is truthful about its demands and donates its resources when its demand is less than its fair share. A non-conformant user, on the other hand, always asks for the maximum of its demand or its fair share (that is, it over-reports its demand during some quanta).

**Resource utilization and system-wide performance improve with more conformant users.** Figure 7(a) and Figure 7(b) show that Karma’s system-wide utilization and performance improve as the fraction of conformant users increases. This is because as more users donate resources when they do not need them, other users can use these resources, improving overall utilization and performance. When none of the users are conformant, since no one ever donates any resources, Karma essentially reduces to strict partitioning, hence achieving low overall utilization and performance. When all users are conformant, Karma achieves optimal utilization and performance, similar to classic max-min fairness.

**Becoming conformant improves user welfare.** Figure 7(c) shows the average welfare gain non-conformant users would achieve if they were to become conformant. When non-conformant users become conformant, it leads to significant ( $1.17\text{--}1.6\times$ ) welfare gains for them, empirically

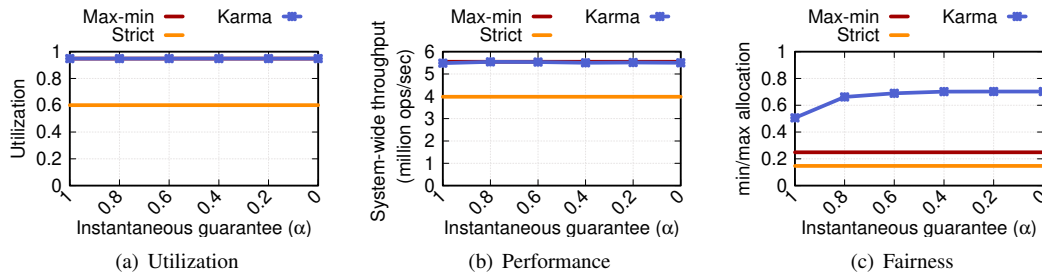


Figure 8: **Sensitivity analysis with varying instantaneous guarantee ( $\alpha$ )** (a, b) Karma matches the resource utilization and system-wide performance of max-min fairness independent of  $\alpha$  (c) Smaller values of  $\alpha$  result in improved long-term fairness.

validating Karma’s property that users have nothing to gain by over-reporting their demand (§3.3). Note that the gain varies with the number of conformant users in the system—the gains from non-conformant users becoming conformant are higher when the percentage of conformant users is low. As expected, the gains show diminishing returns as more users in the system become conformant as overall utilization is already high.

### 5.3 Karma Sensitivity Analysis

We now show sensitivity analysis with the only parameter in the Karma algorithm—the instantaneous guarantee ( $\alpha$ ). Figure 8 shows the resource utilization, system-wide performance, and fairness with  $\alpha$  varying between 0 and 1. Karma continues to match the resource utilization and system-wide performance of max-min fairness independent of  $\alpha$  (Figure 8(a) and Figure 8(b)). Varying  $\alpha$  has an impact on the long-term fairness achieved by Karma (Figure 8(c)), with smaller values of  $\alpha$  resulting in improved fairness, thus validating our discussion in §3.4. Even for  $\alpha = 1$ , Karma is able to achieve significantly better fairness compared to max-min fairness. This is because, while it allocates resources up to the fair share identically to max-min fairness, it prioritizes allocation beyond the fair share based on credits.

## 6 Related Work

There is a large and active body of work on resource allocation and scheduling, exploring various models and settings; it would be a futile attempt to compare Karma with each individual work. We do not know of any other resource allocation mechanism that guarantees Pareto efficiency, strategy-proofness, and fairness similar to Karma for the case of dynamic user demands; nevertheless, we discuss below the most closely related works.

**Max-min fairness variants in cloud resource allocation and cluster scheduling.** Many works study variants of max-min fairness for cloud resource allocation and cluster scheduling [8, 10, 17, 30–33, 44, 46, 57–59, 64, 66, 77], including recent work on ML job scheduling [15, 34, 47, 50, 55]. We make three important notes here. First, while dominant resource fairness

(DRF) [30] has generalized max-min fairness to multiple resources, it makes the same assumptions as max-min fairness: user demands being static over time; our goals are different: we have identified and resolved the problems with max-min fairness for the case of a single resource but over dynamic user demands. It is an interesting open problem to generalize Karma for the case of multiple resources.

Second, cluster scheduling has been studied under several metrics beyond fair resource allocation (*e.g.*, job completion time, data locality, priorities, etc.). Themis [47] considers long-term fairness but defines a new ML workload-specific notion of fairness, and is therefore not directly comparable to Karma. Our goals are most aligned with those works that study fair allocation under strategic users while guaranteeing Pareto efficiency. To that end, the closest to Karma is CARBYNE [32]. However, CARBYNE not only assumes non-strategic users but also, for the single-resource case (the focus of this paper), CARBYNE converges to max-min fairness. As discussed earlier, generalizing Karma to multiple resources remains an open problem; a solution for that problem must be compared against CARBYNE.

Finally, fairness in application-perceived performance is only indirectly related to fairness in resource allocation: other factors like software systems (*e.g.*, hypervisors and storage systems) and resource preemption granularity can impact performance. Similar to other mechanisms [9, 10, 16, 30–33, 39, 45, 46, 59, 60, 66, 67, 72, 77, 80], Karma’s properties are independent of these system-level factors; while our evaluation shows that Karma properties translate to application-level benefits, absolute numbers depend on the underlying system implementation.

**Allocation of time-shared resources.** Generalized Processor Sharing (GPS) [54] is an idealized algorithm for sharing a network link which assumes that traffic is infinitesimally divisible (fluid model). For equal-sized packets and equal flow weights, GPS reduces to Uniform Processor Sharing [54, Section 2], which is equivalent to max-min fairness. GPS guarantees fairness over arbitrary time intervals only under the assumption that flows are *continuously backlogged* [54, Section 2]. This assumption implies that



flows always have demand greater than their fair share, making it trivial to guarantee a max-min fair share of the network bandwidth over arbitrary time intervals. Classical fair-queueing algorithms [11, 24, 48, 65, 83] in computer networks approximate GPS with the constraint of packet-by-packet scheduling. Under this constraint, varying-sized packets and different flow weights make it hard to realize fairness efficiently; thus, the technical question that these algorithms solve is to achieve fairness approximately equal to GPS with minimal complexity. Karma focuses on a different problem—we show that GPS guarantees (equivalent to max-min fairness) are not sufficient when demands are dynamic and present new mechanisms to achieve fairness while maintaining other properties for such dynamic demands.

Stride [74] scheduling essentially approximates GPS in the context of CPU scheduling [74, Section 7], and thus the above discussion applies to it as well. DRF-Q [29] generalizes DRF to support both space and time-shared resources, but is explicitly designed to be memoryless similar to max-min fairness, and therefore suffers from similar issues for long-term fairness. Least Attained Service (LAS) [13, 43, 53] is a classical job scheduling algorithm that has been applied to packet scheduling [13], GPU cluster scheduling [34], and memory controller scheduling [42]. For  $\alpha = 0$ , Karma behaves similarly to LAS, and for  $\alpha > 0$ , Karma generalizes LAS with instantaneous guarantees. Moreover, our results from §3.3 establish strategy-proofness properties of LAS for dynamic user demands, which may be of independent interest.

**Theory works.** Several recent papers in the theory community study the problem of resource allocation for dynamic user demands. Freeman et al. [26] and Hossain et al. [37] consider dynamic demands under a different setting, where users can benefit when they are allocated resources above their demand; under this setting, they focus on instantaneous fairness (which is non-trivial since users can be allocated resources beyond their demand). Karma instead focuses on long-term fairness under the traditional model, where users do not benefit from resources beyond their demands. Sadok et al. [62] present minor improvements over max-min fairness for dynamic demands. Their mechanism allocates resources in a strategy-proof manner according to max-min fairness while marginally penalizing users with larger past allocations using a parameter  $\delta \in [0, 1)$ . For both  $\delta = 0$  and  $\delta \rightarrow 1$ , the penalty goes to 0 for every past allocation, and the mechanism becomes identical to max-min fairness; for other values of  $\delta$ , the penalty is at most a  $\delta(1 - \delta) \leq 1/4$  fraction of past allocation surplus, and it reduces exponentially with time (users who were allocated large amounts of resources further in the past receive an even smaller penalty). Thus, for all values of  $\delta$ , and in particular, for  $\delta = 0$  and  $\delta \rightarrow 1$ , their mechanism suffers from the same problems as max-min fairness. Aleksandrov et al. [7] and Zeng et al. [82] consider dynamic demands, but in a significantly different setting than ours where resources arrive over time.

**Pricing- and credit-based resource allocation.** Another stream of work related to Karma is pricing-based and bidding-based mechanisms for resource allocation, *e.g.*, spot instance marketplace and virtual machine auctions [1, 6, 27, 76, 84, 85]. While interesting, this line of work does not focus on fair resource allocation and is not applicable to use cases that Karma targets. XChange [75] proposes a market-based approach to fair resource allocation in multi-core architectures but focuses on instantaneous fairness rather than long-term fairness, unlike Karma. It assigns a “budget” of virtual currency to each user which can be used to bid for resources. This budget is however reset during every time quantum, and therefore information about past allocations is not carried over.

Credits are used in many other game theoretic contexts [25, 51, 61], *e.g.*, in peer-to-peer and cooperative caching settings to incentivize good behavior among participants with static demands [21, 56, 78]. However, we are not aware of any credit-based mechanisms that deal with resource allocation in the context of dynamic user demands.

## 7 Conclusion

This paper builds upon the observation that the classical max-min fairness algorithm for resource allocation loses one or more of its desirable properties—Pareto efficiency, strategy-proofness, and/or fairness—for the realistic case of dynamic user demands. We present Karma, a new resource allocation mechanism for dynamic user demands, and theoretically establish Karma guarantees related to Pareto efficiency, strategy-proofness, and fairness for dynamic user demands. Experimental evaluation of a realization of Karma in a multi-tenant elastic memory system demonstrates that Karma’s theoretical properties translate well into practice: it reduces application-level performance disparity by as much as  $2.4\times$  when compared to max-min fairness while maintaining high resource utilization and system-wide performance.

Karma opens several exciting avenues for future research. These include (but are not limited to) extending Karma theoretical analysis for  $\alpha > 0$ , generalizing Karma to allocate multiple resource types (similar to DRF), extending Karma to handle all-or-nothing or gang-scheduling constraints which are prevalent in the context of GPU resource allocation [15, 47], and applying Karma to other use cases such as inter-datacenter network bandwidth allocation and resource allocation for burstable VMs in the cloud.

## Acknowledgements

We thank our shepherd, Sebastian Angel, and the OSDI reviewers for their insightful feedback. This research was supported in part by NSF CNS-1704742, CNS-2047220, CNS-2047283, CNS-2104292, CNS-2143868, AFOSR grants FA9550-19-1-0183, FA9550-23-1-0068, a NetApp Faculty Fellowship, an NDSEG fellowship, a Sloan fellowship, and gifts from Samsung, VMware, and Enfabrica.

## References

- [1] Amazon EC2 Spot Instances. <https://aws.amazon.com/ec2/spot/>.
- [2] B-series Burstable Virtual Machine Sizes. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable>.
- [3] CFS: Completely Fair Process Scheduling in Linux. <https://opensource.com/article/19/2/fair-scheduling-linux>.
- [4] Key Concepts and Definitions for Burstable Performance Instances. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html>.
- [5] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting Wide-Area Network Topologies to Solve Flow Problems Quickly. In *NSDI*, 2021.
- [6] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-Driven Memory Allocation. In *VEE*, 2014.
- [7] Martin Aleksandrov and Toby Walsh. Strategy-proofness, Envy-freeness and Pareto efficiency in Online Fair Division with Additive Utilities. In *PRICAI*, 2019.
- [8] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.
- [9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.
- [10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.
- [11] Jon CR Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. *Transactions on Networking*, 1997.
- [12] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosf, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *NSDI*, 2020.
- [13] Ernst W Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller. Scheduling in practice. *Performance Evaluation Review*.
- [14] Qizhe Cai, Midhul Vuppapapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards  $\mu$ s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM*, 2022.
- [15] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys*, 2020.
- [16] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing Alibaba's Co-located Datacenter Workloads. In *Big Data*, 2018.
- [17] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*, 2016.
- [18] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *NSDI*, 2016.
- [19] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a Dynamic Multi-tenant Key-value Cache. In *ATC*, 2017.
- [20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.
- [21] Landon P Cox and Brian D Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *Operating Systems Review*, 2003.
- [22] Alexander D'Amour, Hansa Srinivasan, James Atwood, Pallavi Baljekar, D Sculley, and Yoni Halpern. Fairness is Not Static: Deeper Understanding of Long Term Fairness via Simulation Studies. In *FACCT*, 2020.
- [23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2008.
- [24] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.
- [25] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century Vol 1: Algorithms and Complexity Vol 2: Formal Models and Semantics*. 2004.
- [26] Rupert Freeman, Seyed Majid Zahedi, Vincent Conitzer, and Benjamin C. Lee. Dynamic Proportional Sharing: A Game-Theoretic Approach. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

- [27] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *ATC*, 2016.
- [28] Yingqiang Ge, Shuchang Liu, Ruoyuan Gao, Yikun Xian, Yunqi Li, Xiangyu Zhao, Changhua Pei, Fei Sun, Junfeng Ge, Wenwu Ou, and Yongfeng Zhang. Towards Long-Term Fairness in Recommendation. In *WSDM*, 2021.
- [29] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *SIGCOMM*, 2012.
- [30] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.
- [31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.
- [32] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*, 2016.
- [33] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *OSDI*, 2016.
- [34] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.
- [35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
- [36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.
- [37] Ridi Hossain. Sharing is Caring: Dynamic Mechanism for Shared Resource Ownership. In *AAMAS*, 2019.
- [38] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for  $\mu$ s Latency and High Throughput. In *OSDI*, 2021.
- [39] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.
- [40] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.
- [41] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *EuroSys*, 2022.
- [42] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.
- [43] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory*. 1975.
- [44] Haikun Liu and Bingsheng He. Reciprocal Resource Fairness: Towards Cooperative Multiple-Resource Fair Sharing in IaaS Clouds. In *SC*, 2014.
- [45] Chengzi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Big Data*, 2017.
- [46] Tao Luo, Mingen Pan, Pierre Tholoni, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy Budget Scheduling. In *OSDI*, 2021.
- [47] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*, 2020.
- [48] Paul E McKenney. Stochastic Fairness Queueing. In *INFOCOM*, 1990.
- [49] Deepak Narayanan, Fiodar Kazhemiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *SOSP*, 2021.
- [50] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhemiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.
- [51] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 2001.
- [52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

- [53] Misja Nuyens and Adam Wierman. The Foreground–Background Queue: A Survey. *Performance Evaluation*, 2008.
- [54] Abhay K Parekh and Robert G Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *Transaction on Networking*, 1993.
- [55] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler For Deep Learning Clusters. In *EuroSys*, 2018.
- [56] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do Incentives Build Robustness in BitTorrent. In *NSDI*, 2007.
- [57] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.
- [58] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.
- [59] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *NSDI*, 2016.
- [60] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SoCC*, 2012.
- [61] Tim Roughgarden. Algorithmic Game Theory. *Communications of the ACM*, 2010.
- [62] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. Stateful DRF: Considering the Past in a Multi-Resource Allocation. *Transactions on Computers*, 2021.
- [63] Mohammad Shahradd, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Riccardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. <https://arxiv.org/abs/2003.03423>.
- [64] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, 2010.
- [65] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *SIGCOMM*, 1995.
- [66] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.
- [67] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *RTSS*, 1996.
- [68] Shanjiang Tang, Bu-Sung Lee, Bingsheng He, and Haikun Liu. Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems. In *ICS*, 2014.
- [69] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.
- [70] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *EuroSys*, 2015.
- [71] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. Karma: Resource Allocation for Dynamic Demands. <https://arxiv.org/abs/2305.17222>.
- [72] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an Elastic Query Engine on Disaggregated Storage. In *NSDI*, 2020.
- [73] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.
- [74] Carl A Waldspurger and William E Weihl. *Stride Scheduling: Deterministic Proportional Share Resource Management*. 1995.
- [75] Xiaodong Wang and José F Martínez. XChange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. In *HPCA*, 2015.
- [76] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *SC*, 2017.
- [77] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.



- [78] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *MSSST*, 2013.
- [79] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *OSDI*, 2020.
- [80] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.
- [81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.
- [82] David Zeng and Alexandros Psomas. Fairness-Efficiency Tradeoffs in Dynamic Fair Division. In *EC*, 2020.
- [83] Hui Zhang and Jon CR Bennett. WF2Q: Worst-Case Fair Weighted Fair Queueing. In *INFOCOM*, 1996.
- [84] Liang Zheng, Carlee Joe-Wong, Christopher G Brinton, Chee Wei Tan, Sangtae Ha, and Mung Chiang. On the Viability of a Cloud Virtual Service Provider. In *SIGMETRICS*, 2016.
- [85] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to Bid the Cloud. In *SIGCOMM*, 2015.

# AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

Zhuohan Li<sup>1,\*</sup> Lianmin Zheng<sup>1,\*</sup> Yinmin Zhong<sup>2,\*</sup> Vincent Liu<sup>3</sup> Ying Sheng<sup>4</sup>  
 Xin Jin<sup>2</sup> Yanping Huang<sup>5</sup> Zhifeng Chen<sup>5</sup> Hao Zhang<sup>6</sup> Joseph E. Gonzalez<sup>1</sup> Ion Stoica<sup>1</sup>

<sup>1</sup>UC Berkeley <sup>2</sup>Peking University <sup>3</sup>University of Pennsylvania  
<sup>4</sup>Stanford University <sup>5</sup>Google <sup>6</sup>UC San Diego

## Abstract

Model parallelism is conventionally viewed as a method to scale a single large deep learning model beyond the memory limits of a single device. In this paper, we demonstrate that model parallelism can be additionally used for the statistical multiplexing of multiple devices when serving multiple models, even when a single model can fit into a single device. Our work reveals a fundamental trade-off between the overhead introduced by model parallelism and the opportunity to exploit statistical multiplexing to reduce serving latency in the presence of bursty workloads. We explore the new trade-off space and present a novel serving system, AlpaServe, that determines an efficient strategy for placing and parallelizing collections of large deep learning models across a distributed cluster. Evaluation results on production workloads show that AlpaServe can process requests at up to 10× higher rates or 6× more burstiness while staying within latency constraints for more than 99% of requests.

## 1 Introduction

Advances in self-supervised learning have enabled exponential scaling in model sizes. For example, large pretrained models like BERT [14] and GPT-3 [5] have unlocked a plethora of new machine learning (ML) applications from Copilot [18] to copy.ai [7] and ChatGPT [35].

Serving these very large models is challenging because of their high computational and memory requirements. For example, GPT-3 requires 325 GB of memory to store its parameters as well as a requisite amount of computation to run inference. To serve this model, one would need at least 5 of Nvidia’s newest Hopper 80 GB GPUs just to hold the weights and potentially many more to run in real-time. Worse yet, the explosive growth of model sizes continues unabated [6, 17]. Techniques like model compression and pruning are not sufficient in face of the exponential growth in model sizes and often come at the expense of reduced model quality [15].

\*Equal contribution.

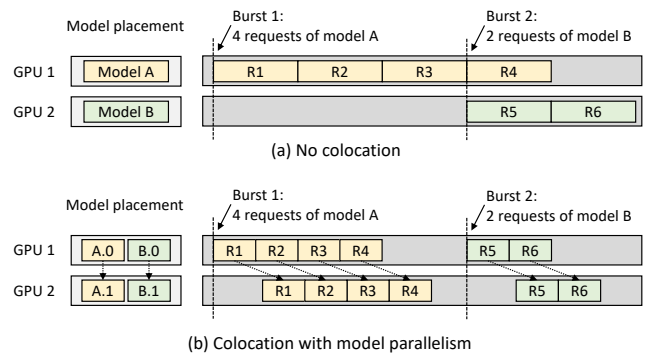


Figure 1: Two placement strategies for serving two models on two GPUs. In each subfigure, the left part shows the model placements and the right part shows the timeline for handling bursty requests. At the time of "Burst 1", 4 requests of model A come at the same time. Collocation with model parallelism can reduce the average completion time of bursty requests.

Provisioning sufficient resources to serve these models can be arduous as request rates are bursty. For example, using common workload traces, we observe frequent spikes in demand of up to 50× the average [54]. Meeting the service level objective (SLO) of latency usually means provisioning for these peak loads, which can be very expensive; additional devices allocated for this purpose would remain underutilized most of the time. Making matters worse, it is increasingly common to serve multiple models and multiple variations of the same large model in situations like A/B testing or serving fine-tuned models for specific domains (§2).

This paper studies how to efficiently serve multiple large models concurrently. Specifically, we explore the underappreciated benefits of model parallelism in online model serving, even for smaller models that can fit on a single device. Model parallelism refers to partitioning and executing a model on distributed devices (§2.1). The benefits of model parallelism have been well studied [23, 27, 31, 56] in the *throughput-oriented* training setting. However, its effects for model serving under *latency-sensitive* settings remains largely untapped.

We observe that there are fundamental transition points in

the model serving design space that challenge prior assumptions about serving, even for models that fit on a single device. For example, consider the scenario with two models and two GPUs, each of which has sufficient memory to hold one complete model. As shown in Fig. 1(a), the natural approach assumed by almost all existing serving systems [9, 33, 34] is to allocate one dedicated GPU for one model. This approach appears rational because partitioning the model across GPUs would incur communication overheads that would likely increase the prediction latency. However, we find that inducing additional model parallelism (to the point where per-example execution time actually *increases*) enables a wider range of placement strategies, e.g., model co-location, which can improve the statistical multiplexing of the system under bursty workloads. In Fig. 1(a), assuming the execution time of a model is  $y$ , the average end-to-end latency of request 1 through 4 is  $(1y + 2y + 3y + 4y)/4 = 2.5y$ . In Fig. 1(b), assuming a 10% model-parallel overhead, the average latency of request 1 through 4 is reduced to  $(1.1y + 1.6y + 2.1y + 2.6y)/4 = 1.85y$ . Co-location with model parallelism can utilize more devices to handle bursty requests and reduces the average completion time, despite its overheads (§3.1). Even if we batch the requests, the case still holds (§6.5).

Unfortunately, the decision of how to optimally split and place a collection of models is complex. Although leveraging model parallelism as above has its benefits, it still adds overheads that may negate those benefits for less bursty workloads. For example, we find that a particularly influential axis on the efficacy of model parallelism is per-GPU memory capacity (§3.2), although other factors (e.g., the arrival pattern, SLO) can also have a significant effect. Further, besides the inter-op model parallelism presented in Fig. 1, another kind of model parallelism, intra-op parallelism, presents its own distinct tradeoffs (§3.3). Ultimately, different styles of parallelism and their tradeoffs create a complex, multi-dimensional, and multi-objective design space that existing systems largely ignore and/or fail to navigate. However, not leveraging model parallelism in the serving setting is typically not an option for large models, and not addressing this trade-off space directly results in significant increases in cost and serving latency.

To that end, we present AlpaServe<sup>2</sup>, a system that automatically and efficiently explores the tradeoffs among different parallelization and placement strategies for model serving. AlpaServe takes a cluster resource specification, a set of models, and a periodic workload profile; it then partitions and places the models and schedules the requests to optimize SLO attainment (i.e., the percentage of requests served within SLO). To assist the design of AlpaServe, we first introduce a taxonomy and quantify the tradeoffs between different parallelization strategies in model serving (§3). We then present key algorithms to navigate the tradeoff space (§4). We design an iterative simulator-guided model placement algorithm

to optimize the collocation of models and a group partition algorithm to search for the best way to partition the cluster into disjoint model-parallel groups. In addition, we extend the existing auto-parallelization algorithms for training to make them more suitable for inference.

We evaluate AlpaServe with production workloads on a 64-GPU cluster (§6). Evaluation results show that, when optimizing one metric at a time, AlpaServe can choose to increase the request processing rate by  $10\times$ , achieve  $2.5\times$  lower latency deadlines, or tolerate  $6\times$  burstier traffic compared to previous state-of-the-art serving systems.

In summary, we make the following contributions:

- A detailed analysis of the tradeoff space of different model parallel strategies for efficient model serving.
- Novel model placement algorithms to incorporate model parallelism in a serving system.
- A comprehensive evaluation of AlpaServe with both synthetic and production workloads.

## 2 Background

Over the past few years, increasingly capable models have been developed for everything from recommendations to text generation. As a result, serving predictions from these models has become an essential workload in modern cloud systems. The structure of these workloads often follows a simple request-response paradigm. Developers upload a pre-trained model and its weights ahead of time; at runtime, clients (either users or other applications) submit requests for that model to a serving system, which will queue the requests, dispatch them to available GPUs/TPUs, and return the results.

The requirements of these model-serving systems can be stringent. To satisfy user demand, systems often must adhere to aggressive SLO on latency. At the same time, serving systems that must run continuously need to minimize their operational costs associated with expensive accelerators. Minimizing serving costs can be challenging because dynamically scaling compute resources would be too slow on the critical path of each prediction request: it can take multiple seconds just to swap a large model into accelerator memory [37]. Furthermore, there is significant and unpredictable burstiness in the arrival process of user requests. To meet tight SLO, contemporary serving systems are forced to over-provision compute resources, resulting in low cluster utilization [48].

Another pattern that emerges in serving large models is the use of multiple instances of the same or similar model architectures. This is commonly seen in the practice of pretraining on large unlabeled data and fine-tuning for various downstream tasks [14], which can significantly boost accuracy but results in multiple instances of the same model architecture. For example, Hugging Face serves more than 9,000 versions of fine-tuned BERT [24]. They either share a portion of the parameters or do not share any parameters at all for better accuracy. Prior works have [44, 57] exploited the property of shared parameters, but we do not consider the shared parame-

<sup>2</sup><https://github.com/alpa-projects/mms>

ters in this paper because AlpaServe targets general settings and full-weight tuning is still a major use case.

## 2.1 Model Parallelism in Model Serving

Distributed parallel model execution is necessary when attempting to satisfy the serving performance requirements or support large models that do not fit in the memory of a single device. At a high level, distributed execution of deep learning models can be classified into two categories: intra-operator parallelism and inter-operator parallelism [56].

**Intra-operator parallelism.** DL models are composed of a series of operators over multidimensional tensors, e.g., matrix multiplication over input and weight tensors. Intra-operator parallelism is when a single operator is partitioned across multiple devices, with each device executing a portion of the computation in parallel [43, 45, 50]. Depending on the specific partitioning strategy and its relationship to prior and subsequent operators in the model, partitioning can require communication among participating GPUs to split the input and then merge the output.

The benefit of intra-operator parallelism for single-request execution is twofold. First, it can expand the total amount of computation available to the target model, reducing its end-to-end latency. In a similar fashion, it can expand the total memory available to the model for storing its inputs, weights, and intermediate values. The cost is the aforementioned communication overhead.

**Inter-operator parallelism.** The other type of parallelism available to DL models is inter-operator parallelism, which assigns different operators of the model’s execution graph to execute on distributed devices in a pipeline fashion (a.k.a. pipeline parallelism) [23, 28, 30]. Here, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs.

Unlike intra-operator parallelism, pipeline parallelism does not reduce the execution time of a single request. In fact, it typically increases the execution time due to modest amounts of communication latency between pipeline stages, although the total amount of transferred data is often lower than it is in intra-operator parallelism. Instead, the primary use of inter-operator parallelism in traditional serving systems is to allow the model to exceed the memory limitation of a single GPU.

## 3 Motivation and Tradeoff Analysis

As mentioned, both types of model parallelism reduce per-device memory usage by partitioning a model on multiple devices. A key motivation for this work is that we can use this property to fit more models on one device, enabling better statistical multiplexing of the devices when handling bursty requests. We explore this idea through a series of empirical examinations and theoretical analysis, starting with an illustrative example (§3.1), followed by an empirical analysis of when model parallelism is beneficial (§3.2), the overhead of

model parallelism (§3.3), and a queueing theory-based analysis (§3.4). All the experiments in this section are performed on an AWS EC2 p3.16xlarge instance with 8 NVIDIA 16GB V100 GPUs.

### 3.1 Case Study: A Two-model Example

We start with an illustrative experiment to show how model parallelism can benefit the serving of multiple models. We use two GPUs to serve two Transformer models with 6.7 billion parameters each (13.4 GB to store its FP16 weights). Because each GPU has 16 GB of memory, it can fit one and only one model. A single request takes around 0.4 s to process on one GPU.

We compare the following model placements, corresponding to the strategies in Fig. 1. The first is *simple placement*, where we place one model on each GPU due to the memory constraint. The second is *model-parallel placement*, where we use inter-op parallelism to partition each model to a 2-stage pipeline and let each GPU execute half of each model.

We evaluate the two placements when the requests to each model follow an independent Poisson process with an arrival rate of 1.5 request/s. Fig. 2a shows the cumulative distribution function (CDF) and average of request latency (which includes the GPU execution time and queuing delay). Model-parallel placement reduces the average latency of the simple placement from 0.70s to 0.55s, a  $1.3\times$  speedup. The speedup comes from the better burst tolerance: when a burst arrives that exceeds the capability of a single GPU, simple placement must begin queuing requests. However, as long as the other model does not receive many requests, the model parallel placement can use both GPUs to serve the requests for the popular model via statistical multiplexing of the GPUs.

This effect becomes more pronounced with higher burstiness, which we can demonstrate using a Gamma request arrival process with the same average request rate as above but a higher coefficient of variance (CV) of 3. As shown in Fig. 2b, the speedup on mean latency is now increased to  $1.9\times$ . Fig. 2d shows a representative trace of the corresponding total cluster utilization over time. Note that for each request burst, model-parallel placement can use the whole cluster and only take half of the time to process, while simple placement can only use half of the cluster.

In addition, we also evaluate the case where one model receives more requests than another. In Fig. 2c, we use Poisson arrival but let 20% of the requests ask for model 1 and 80% ask for model 2. Although replication performs slightly better for model 1 requests, it is drastically worse on model 2 requests compared to the model-parallel placement. For model-parallel placement, because both GPUs are shared across two models, the requests to both models follow the same latency distribution. Overall, model-parallel placement reduces the mean latency by  $6.6\times$ .



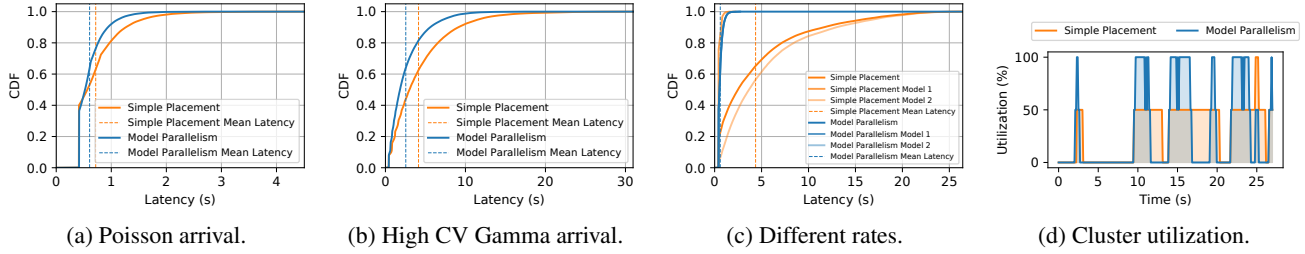


Figure 2: Latency CDF and cluster utilization in the 2-model example.

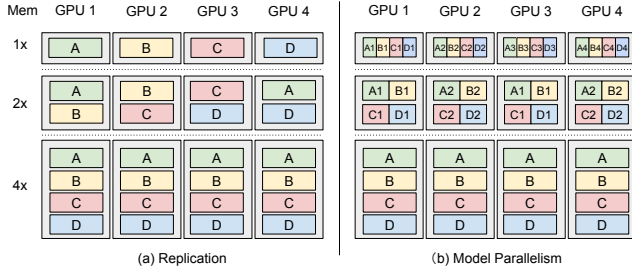


Figure 3: Replication and model parallel placement illustration with different memory budgets, where the memory budgets are set to be multiples of a single model’s size.

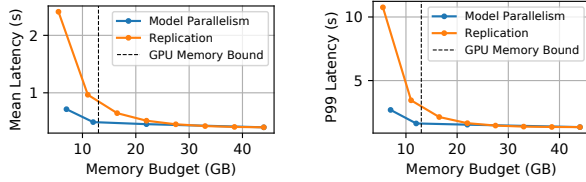


Figure 4: Serving performance with changing per-GPU memory budgets. Model parallelism is beneficial for limited memory budget. The dashed vertical line is the real per-GPU memory bound of a 16GB V100. The value is around 13GB due to the need to store activations and other runtime context.

### 3.2 When is Model Parallelism Beneficial

To further explore the nuances of model parallelism in serving, we increase the size of the deployment to 8 GPUs and 8 Transformer models with 2.6B parameters each. As a base setting, we set the requests to each model as a Gamma process with an average rate of 20 request/s and CV of 3; we then vary a range of factors to see their effects. Note that some of the settings we evaluate are impossible on real hardware (e.g., exceeding the memory capacity of a single device) so we leverage the simulator introduced in §5. The fidelity of the simulator is very high as verified in §6.1.

The model in this case is smaller (5.2GB), so one GPU can also store multiple models without model parallelism. We compare two placement methods: (1) *Replication*. In this setting, we replicate the models to different devices until each device cannot hold any extra models. Because all the models receive equal amounts of loads on average, we replicate each model the same number of times (Fig. 3a). (2) *Model Parallelism*.

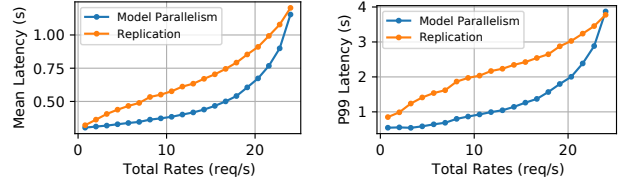


Figure 5: Serving performance with changing arrival rates. Model parallelism is beneficial for smaller rates.

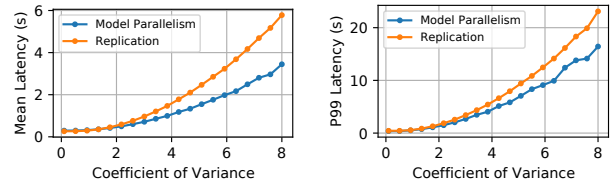


Figure 6: Serving performance with changing CVs. Model parallelism is beneficial for larger CVs.

*lism*. Here we use inter-operator parallelism and uniformly assign the Transformer layers to different GPUs.

**Device memory.** We evaluate the mean and the tail latency of the two placement methods under different device memory capacities. For replication, more GPU memory can fit more models onto a single GPU. For model parallelism, more GPU memory can also reduce the number of pipeline stages and reduce the overhead as in Fig. 3b. The resulting mean and P99 latency is shown in Fig. 4. With more memory, more models can fit into a single GPU, so the benefit of statistical multiplexing diminishes because replication can also effectively use multiple devices to serve the bursty requests to a single model. When the GPU memory capacity is large enough to hold all models, there is no gain from model parallelism.

**Request arrival.** We vary the parameters of the arrival process and compare the replication placement with the model-parallel placement with 8-stage pipeline parallelism. The mean and P99 latency results of changing arrival rate are shown in Fig. 5. When the arrival rate is low, model parallelism can greatly reduce the serving latency. However, when the arrival rate approaches the peak serving rate of the cluster, the benefit of model-parallel placement starts to diminish. Eventually, it starts to perform worse than replication. This is because when all models are equally saturated, the replication

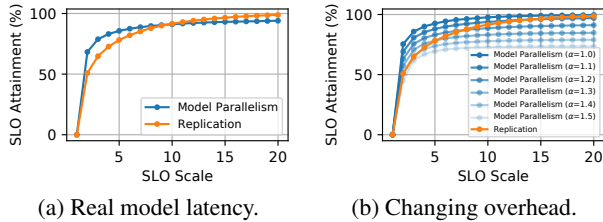


Figure 7: SLO attainment with changing SLOs. Model parallelism is beneficial for smaller SLOs.

placement is able to achieve efficient cluster utilization and there is no benefit to the statistical multiplexing afforded by model parallelism. Instead, the overhead of model parallelism (§3.3) starts to become a significant factor.

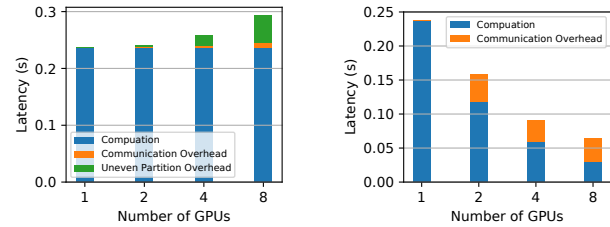
The mean and P99 latency results of changing CV are in Fig. 6. With a higher CV, the requests become more bursty, and the benefit of model parallelism becomes more significant. As shown in the results, with a higher CV, model parallelism can greatly outperform the performance of replication.

**Service level objectives.** In prediction serving settings, it is common to have tight latency SLO and predictions made after these deadlines are often discarded [19]. For example, advertising systems may choose not to show an ad rather than delay rendering user content. In this case, the goal of the serving system is to optimize the percentage of requests that can be finished within the deadline, i.e., *SLO attainment*.

In this experiment, we measure how SLOs affect the performance of the placement methods. We compare the replication and the model-parallel placement with 8-stage pipeline parallelism. During execution, we drop the requests that will exceed the deadline even if we schedule it immediately. We scale the SLO to different multiplies of the single device execution latency (*SLO Scale* in Fig. 7a) and compare the SLO attainment of the two methods.

As in Fig. 7a, when SLO is tight ( $< 10\times$  model latency), model parallelism can greatly improve SLO attainment. However, when the SLO becomes looser, its SLO attainment plateaus but that of the replication placement keeps growing. This result shares the same core logic as previous experiments: When SLO becomes looser, more requests can stay in the waiting queue, and thus the effective burstiness of the requests decreases. When many requests are queued, the system is bounded by its total processing capability, which might be affected by the model parallelism overhead. In the real world, the SLO requirement is often less than  $5\times$  of the model execution latency [19], where model parallelism can improve SLO attainment.

**Summary:** Model parallelism benefits model serving through statistical multiplexing when the device memory is limited, the request rate is low, the request CV is high, or the SLO is tight.



(a) Inter-op parallelism. (b) Intra-op parallelism.

Figure 8: The overhead decomposition. The overhead of inter-op parallelism mainly comes from uneven partition while the overhead of intra-op parallelism comes from communication.

### 3.3 Overhead of Model Parallelism

In this section, we further investigate the overheads of different model parallel strategies and how they affect serving performance. Similar to the setup in Fig. 7a, we manually modify the overhead of model parallelism. Specifically, let the latency of a single model executing on the GPU be  $L$  and the number of pipeline stages be  $n$ . We set the total latency of pipeline execution to be  $\alpha L$  and the latency of each pipeline stage to be  $\alpha L/n$ , where  $\alpha$  is a parameter that controls the overhead. When  $\alpha = 1$ , model parallelism does not have any overhead and larger  $\alpha$  means higher overhead.

We show the results in Fig. 7b. If model parallelism does not have any overhead ( $\alpha = 1$ ), it can always outperform replication due to its ability to multiplex the devices. When the overhead becomes larger and the SLO is low, model parallelism still outperforms replication. However, with a larger SLO, the effective burstiness is reduced and the performance is dominated by the overhead.

Given that the overhead can greatly affect serving performance, we perform a detailed study of the multiple sources of model-parallel overhead in Fig. 8. For inter-op parallelism, when partitioning a single model into multiple stages, different stages need to communicate the intermediate tensors, and we denote this overhead as the *communication overhead*. In addition, the pipeline execution will be bottlenecked by the execution time of the slowest stage, making the effective latency to be the number of pipeline stages times the latency of the slowest stage [23]. We denote this as the *uneven partition overhead*. As in Fig. 8a, for inter-op parallelism, most overhead comes from the latency imbalance among different pipeline stages, instead of the communication between stages. While our previous discussion mainly focuses on inter-op parallelism, the other type of model parallelism, intra-op parallelism, has very different performance characteristics. Its overhead is merely brought by the collective communication across multiple devices [31], which cannot be overlapped with the neural network computation due to data dependency. From Fig. 8b, we can see that the communication overhead of intra-op parallelism is much higher than inter-op parallelism.

Finally, we compare the latency, throughput, and memory consumption of different model-parallel placements and the

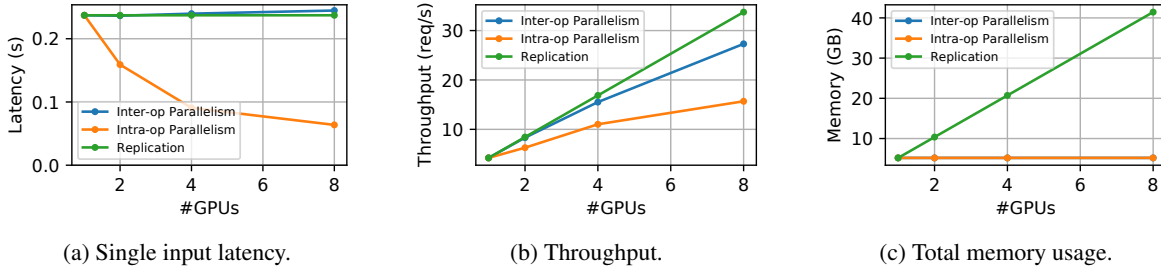


Figure 9: The latency, throughput and memory usage vs. #GPUs for inter-op parallelism, intra-op parallelism, and replication. In subfigure (c), the lines for inter-op and intra-op parallelism overlap.

replication placement in Fig. 9. Because of the sequential dependence between the different stages, inter-op parallelism cannot reduce the execution latency of a single input data. Instead, the latency is slightly higher due to the communication between the stages. On the other hand, intra-op parallelism can largely reduce the latency via the parallel execution of different GPUs (Fig. 9a). However, because inter-op parallelism can pipeline the execution of different stages and only communicate a relatively small amount of data, it attains higher throughput compared to intra-op parallelism (Fig. 9b). Because both parallel methods split the model weight tensors across different GPUs, the total memory usage stays constant with increasing numbers of GPUs (Fig. 9c). This makes the statistical multiplexing of different GPUs across multiple models possible.

In the end, the tradeoff between parallelization strategies and their interplay with cluster resources, arrival patterns, and serving objectives forms an intricate design space.

### 3.4 Queuing Theory Analysis

In this section, we use queuing theory to mathematically verify the conclusions in §3.2 and §3.3. Specifically, we analyze the case where the inputs follow the Poisson arrival process. Since the execution time of a deep learning inference task is highly predictable [19], we assume the request serving time is deterministic. For the single device case, suppose the request rate to a model is  $\lambda_0$  and the single device latency is  $D$  with the utilization  $\lambda_0 D < 1$ , then the average number of requests  $L_Q$  and the average latency  $W$  in this M/D/1 queue [46] are:

$$L_Q = \frac{\lambda_0 D}{2(1 - \lambda_0 D)}, \quad W = D + L_Q D = D + \frac{\lambda_0 D^2}{2(1 - \lambda_0 D)}.$$

Now consider the example in §3.1. Let  $p\lambda, (1-p)\lambda$  be the average request rates for the two models respectively, where  $p \in [0, 1]$  controls the percentage of requests for both models. Then for the simple placement, the average latency can be derived as the average latency of two independent queues:

$$W_{simple} = D + \frac{p^2 \lambda D^2}{2(1 - p\lambda D)} + \frac{(1-p)^2 \lambda D^2}{2(1 - (1-p)\lambda D)}.$$

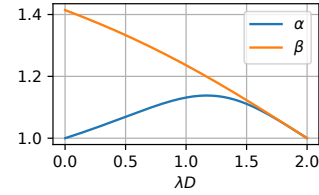


Figure 10: Maximal communication overhead  $\alpha$  and uneven partition overhead  $\beta$  satisfy  $W_{pipeline} \leq W_{simple}$  as a function of total utilization  $\lambda D$ .

Note that  $W_{simple}$  reaches minimum when  $p = 1/2$ . Intuitively, when  $p$  is not exactly half, one model receives more requests than the other. This larger portion of requests have a longer queuing delay, which leads to the higher overall mean latency.

For the model-parallel case, the requests to both models merged to a single Poisson Process with rate  $\lambda$ . For pipeline parallelism, suppose the latency for a single input to be  $D_s$  and the maximum stage latency to be  $D_m$ , then the average latency would be

$$W_{pipeline} = D_s + \frac{\lambda D_m^2}{2(1 - \lambda D_m)}.$$

Suppose there is no model-parallel overhead, then  $D_s = 2D_m = D$ . Let's first consider the case where  $p = 1/2$  (Fig. 2a). We have

$$W_{simple} = D + \frac{\lambda D^2}{4 - 2\lambda D}, \quad W_{pipeline} = D + \frac{\lambda D^2}{8 - 4\lambda D}.$$

In this case, the waiting time for model-parallel execution is half of the simple placement waiting time, as shown in the vertical lines in Fig. 2a. When the  $p$  is not 1/2,  $W_{simple}$  will increase while  $W_{pipeline}$  will stay the same, so the gap between  $W_{simple}$  and  $W_{pipeline}$  will be even larger, as in Fig. 2c.

Next, we consider the case where model parallelism incurs overhead. We measure the two types of overheads in §3.3 separately: With the overhead from communication,  $D_s = 2D_m = \alpha D$ , where  $\alpha \geq 1$  is the overhead factor. With the overhead from uneven stages, we suppose  $D_s = D$  still holds, but  $D_m = \beta D/2$  where  $\beta \geq 1$  is the overhead factor. To keep  $W_{pipeline} \leq W_{simple}$ , we can get the maximal  $\alpha$  and

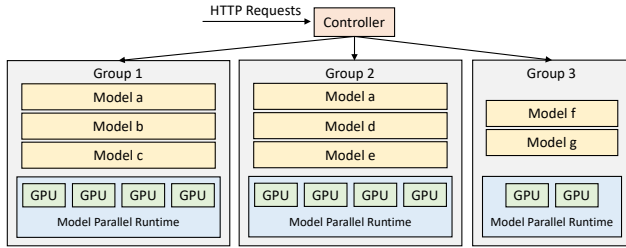


Figure 11: AlpaServe Runtime System Architecture

$\beta$  as a function of the total utilization  $\lambda D$  separately and we visualize the function in Fig. 10. When the utilization is high, the benefit of statistical multiplexing diminishes, and thus the overhead needs to be low, as in §3.2. On the other hand, when the utilization is very low, most requests will not be queued, and thus the communication overhead  $\alpha$  needs to be low to keep the processing latency to be low. Note that the maximal overhead here is based on a uniform Poisson arrival distribution. A more bursty or more non-uniform arrival distribution will make the simple placement performs worse and make the model-parallelism placement outperforms the simple replication placement with even higher overhead.

## 4 Method

From §3, we can see that there are several key challenges to effectively utilize model parallelism for deep learning serving:

- Derive efficient model parallel strategies for inference to reduce the overhead of model parallelism. Specifically, find a partitioning strategy that minimizes the stage imbalance for inter-operator parallelism.
- Determine model-parallel placements according to the arrival pattern to maximize SLO attainment.

We built *AlpaServe* to specifically tackle these challenges. The runtime architecture of AlpaServe is shown in Fig. 11. AlpaServe utilizes a centralized controller to dispatch requests to different groups.<sup>3</sup> Each group hosts several model replicas on a shared model-parallel runtime. This section describes the architecture of AlpaServe and the key algorithms for efficiently leveraging model parallelism in a model serving system.

### 4.1 Automatic Parallelization for Inference

Since different parallelization configurations have different latency and throughput trade-offs, we need to enumerate multiple possible configurations for every single model and let the placement algorithm choose the best combination for all models in the whole cluster. Therefore, given a model, AlpaServe first runs an auto-parallelization compiler with various constraints to generate a list of possible configurations. We build several extensions on top of an existing auto parallelization training system, Alpa [56], to make it suitable for generating serving parallelization strategies. Alpa includes two passes

<sup>3</sup>For a larger service, AlpaServe can be extended as a hierarchical deployment with each controller only managing a subset of devices as in [52].

for generating efficient model parallel partitions: inter-op pass and intra-op pass. The inter-op pass uses a dynamic programming (DP) algorithm to figure out the optimal inter-op parallel plan, and it calls the intra-op pass for each potential pipeline stage, which is formulated as an integer linear programming (ILP) problem, to profile its latency with the optimal intra-op parallel plan. In AlpaServe, we keep the two compilation passes, but extends both passes for serving.

The inter-op pass in Alpa optimizes the overall pipeline execution latency, which includes the time of forward and backward propagation and weight synchronization. However, in serving workloads, only forward propagation is being executed and there is no need for weight synchronization. Therefore, we reformulate the dynamic programming in AlpaServe to merely focus on minimizing the maximal stage latency. Specifically, denote  $F(s, k)$  to be the maximum latency when slicing layers 1 to  $k$  into  $s$  stages. We can derive  $F$  as

$$F(s, k) = \min_{1 \leq i \leq k} \{\max\{F(s-1, i-1), \text{latency}(i, k)\}\},$$

where  $\text{latency}(i, k)$  denotes the latency of a stage composes of layer  $i$  to  $k$ . In Alpa, the  $\text{latency}$  function of all possible  $O(K^2)$  combinations is being profiled by the intra-op pass because of the complicated dependency between forward and backward passes. In AlpaServe, because the pipeline stages only perform forward propagation and only communicate intermediate results once between layer boundaries, we can accelerate the profiling by only profiling  $K$  layers and letting  $\text{latency}(i, k)$  to be the sum of the latencies for layer  $i$  to  $k$ . This acceleration enables us to efficiently enumerate different inter- and intra-op device partition setups and generate a list of parallel strategies for the placement algorithm in §4.2.

For the intra-op pass, we extend the ILP in Alpa to drop all configurations that use data parallelism. For serving workloads, because there is no need for weight synchronization, data parallelism can be achieved by the replication placement. We leave the decision of whether to replicate a model to the placement algorithm in §4.2.

### 4.2 Placement Algorithm

Given a set of models and a fixed cluster, AlpaServe partitions the cluster into several groups of devices. Each group of devices selects a subset of models to serve using a shared model-parallel configuration. Different groups can hold the same model as replicas. The requests for a model are dispatched to the groups with the requested model replica. We call a specific cluster group partition, model selection, and parallel configuration as a *placement*. Our goal is to find a placement that maximizes the SLO attainment.

However, finding the optimal placement is a difficult combinatorial optimization problem. The overall placement configuration space grows exponentially with the number of devices and the number of models. To make things worse, the objective “SLO attainment” has no simple analytical formula for



---

**Algorithm 1** Simulator-Guided Greedy Model Selection.

**Input:** Model list  $M$ , device group list  $G$ , group parallel configurations  $P$ , workload  $W$ , beam size  $k$  (default = 1).

**Output:** The model selection  $best\_sel$ .

```
best_sel ← ∅
beam_sels ← {∅}
while true do
  new_sels ← ∅
  for sel ∈ beam_sels do
    for (m, (g, p)) ∈ M × (G, P) do
      // Parallelize the model as in §4.1.
      m_parallelized ← parallelize(m, g, p)
      sel' ← sel.add_model_to_group(m_parallelized, g)
      if sel' is in memory constraint then
        sel'.slo_attainment ← simulate(sel', W)
        new_sels.append(sel')
  if new_sels = ∅ then
    break
  beam_sels ← top-k_SLO_attainment(new_sels)
  sel* ← pick_highest_SLO_attainment(beam_sels)
  if sel*.slo_att > best_sel.slo_att then
    best_sel ← sel*
return best_sel
```

---

an arbitrary arrival distribution. Existing tools and approximations from queueing theory can only analyze simple cases in §3.4 and cannot model more complex situations [46]. Therefore, we resort to a simulator-guided greedy algorithm that calls a simulator to compute SLO attainment.

To compute the SLO attainment with a given set of requests and placement, in AlpaServe, we assume we know the arrival process in advance. Although short-term burstiness is impossible to predict, the arrival pattern over longer timescales (e.g., hours or days) is often predictable [48]. Given this predictability, AlpaServe either directly uses the history request traces or fits a distribution from the trace and resamples new traces from the distribution as the input workload to the simulator to compute the SLO attainment.

We design a two-level placement algorithm: Given a cluster group partition and a shared model-parallel configuration for each group, Algorithm 1 uses a simulator-guided greedy algorithm to decide which models to select for each group. Then, Algorithm 2 enumerates various potential cluster partitions and parallel configurations and compares the SLO attainment from Algorithm 1 to determine the optimal placement.

Given a cluster group partition with a fixed model-parallel configuration for each group, Algorithm 1 selects model replicas iteratively as a beam search algorithm: At each iteration, it enumerates all possible (model, group) pairs, parallelizes the model on the device group with the algorithms in §4.1, and checks whether the model can be put on the group under the memory constraint. For all valid selections, it runs the

---

**Algorithm 2** Enumeration-Based Group Partition and Model-Parallel Configuration Selection.

**Input:** Model list  $M$ , cluster  $C$ , workload  $W$ .

**Output:** The placement  $best\_plm$ .

```
best_plm ← ∅
B ← get_potential_model_buckets(M)
for (B1, B2, ..., Bk) ∈ B do
  H ← get_potential_device_buckets(C, B, k)
  // Get the placement for each bucket individually.
  for i from 1 to k do
    plmi* ← ∅
    G ← get_potential_group_partitions(Hi)
    for G ∈ G do
      P ← get_potential_parallel_configs(G)
      for P ∈ P do
        plm ← greedy_selection(Bi, G, P, W)
        if plm.slo_att > plmi*.slo_att then
          plmi* ← plm
    plm* ← concat(plm1*, ..., plmk*)
  if plm*.slo_att > best_plm.slo_att then
    best_plm ← plm*
return best_plm
```

---

simulator and computes SLO attainment. It then picks the top- $k$  solutions and enters the next iteration. The algorithm terminates when no more replicas can be put into any groups.

The complexity of Algorithm 1 is  $O(MGRS)$ , where  $M$  is the number of models,  $G$  is the number of groups,  $R$  is the number of replicas we can put according to the memory constraint,  $S$  is the number of requests in the workload (the simulation time is proportional to the number of the requests) and  $B$  is the beam size. It runs reasonably fast for our medium-scale cluster when the number of requests is small. When the number of requests is very large, we propose another heuristic to accelerate: Instead of using the simulator to evaluate all (model, group) pairs at each iteration, we can run the simulator only once and place a model with the most unserved requests in an available group with the lowest utilization. This reduces the time complexity to  $O((M + G)RS)$ . We find this heuristic gives solutions with SLO attainment higher than 98% of the SLO attainment get by the original algorithm in our benchmarks.

Algorithm 2 enumerates different group partitions and model-parallel configurations and picks the best one via multiple calls to Algorithm 1. When designing Algorithm 2, the first phenomenon we notice is that putting small and large models in the same group causes convoy effects, where the requests of small models have to wait for the requests of large models and miss the SLO. Therefore, in Algorithm 2, we first cluster models into model buckets. Each bucket contains a set of models with relatively similar sizes

and every model is assigned to one and only one bucket. Specifically, the function `get_potential_model_buckets` returns all the possible model bucket partitions that separate models whose latency difference is larger than a threshold into different disjoint buckets. We then enumerate all the potential ways to assign the devices to each bucket in `get_potential_device_buckets`.

Because different buckets include a disjoint set of models, we can then figure out the optimal placement for each bucket individually. For each bucket, we enumerate possible ways to partition the devices in the bucket into several groups in `get_potential_group_partitions` and enumerate the potential parallel configurations for each group with the method in `get_potential_parallel_configs`. We then call Algorithm 1 with `greedy_placement` to place models in the model bucket to the groups in the device bucket. We send the whole workload  $W$  to Algorithm 1, which ignores the requests that hit the models outside of the current bucket. Finally, a complete solution is got by concatenating the solutions for all buckets. The algorithm returns the best solution it finds during the enumerative search process.

Enumerating all possible choices can be slow, so we use the following heuristics to prune the search space. Intuitively, we want the different buckets to serve a similar number of requests per second. Therefore, we eliminate the bucket configurations with high discrepancies in the estimated number of requests it can serve per second for each bucket. Additionally, in `get_potential_group_partitions` and `get_potential_parallel_configs`, we assume all groups have the same size and the same parallel configurations except for the last group which is used when the number of devices is not divisible by the group size.

### 4.3 Runtime Scheduling

We use a simple policy to dispatch and schedule the requests at runtime. All requests are sent to a centralized controller. The controller dispatches each request to the group with the shortest queue length. Each group manages a first-come-first-serve queue. When a group receives a request, it checks whether it can serve the request under SLO and rejects the request if it cannot. This is possible because the execution time of a DNN model is very predictable and can be got in advance by profiling [19]. In most of our experiments, we do not include advanced runtime policies such as batching [19], swapping, and preemption [21]. These techniques are complementary to model parallelism. Nevertheless, we discuss how they fit into our system.

**Batching.** Batching multiple requests of the same model together can increase the GPU utilization and thus increase the throughput of a serving system. In our system, we do find batching is helpful, but the gain is limited. This is because we mainly target large models and a small batch size can already fully saturate the GPU, which is verified in §6.5. To isolate the benefits of model parallelism and make the results more

explainable, we decide to disable any batching in this paper except for the experiments in §6.5.

**Preemption.** The optimal scheduling decision often depends on future arrivals, and leveraging preemption can help correct previous suboptimal decisions. The first-come-first-serve policy may result in convoy effects when models with significantly different execution times are placed in the same group. We anticipate a least-slack-time-first policy with preemption can alleviate the problems [12].

**Swapping.** The loading overheads from the CPU or Disk to GPU memory are significant for large models, which is the target of this paper, so we do not implement swapping in AlpaServe. We assume all models are placed on the GPUs. This is often required due to tight SLOs and high rates, especially for large models. The placement of models in AlpaServe can be updated in the periodic re-placement (e.g., every 24 hours).

**Fault tolerance.** While the current design of AlpaServe does not have fault tolerance as a focus, we acknowledge several potential new challenges for fault tolerance: With model parallelism, the failure of a single GPU could cause the entire group to malfunction. Additionally, the use of a centralized controller presents a single point of failure.

## 5 Implementation

We implement a real system and a simulator for AlpaServe with about 4k lines of code in Python. The real system is implemented on top of an existing model-parallel training system, Alpa [56]. We extend its auto-parallelization algorithms for inference settings to get the model-parallel strategies. We then launch an Alpa runtime for each group and dispatch requests to these groups via a centralized controller.

The simulator is a continuous-time, discrete-event simulator [39]. The simulator maintains a global clock and simulates all requests and model executions on the cluster. Because the simulator only models discrete events, it is orders of magnitude faster than the real experiments. In our experiment, it takes less than 1 hour for a 24-hour trace. The fidelity of the simulator is very high because of the predictability of DNN model execution, which is verified in §6.1.

## 6 Evaluation

In this section, we evaluate AlpaServe’s serving ability under a variety of model and workload conditions. The evaluation is conducted on a range of model sizes, including those that do and do not fit into a single GPU, and we show that AlpaServe consistently outperforms strong baselines across all model sizes. In addition, we evaluate the robustness of AlpaServe against changing arrival patterns and do ablation studies of our proposed techniques. Evaluation results show that AlpaServe can greatly improve various performance metrics. Specifically, AlpaServe can choose to save up to  $2.3\times$  devices, handle  $10\times$  higher rates,  $6\times$  more burstiness, or  $2.5\times$  more stringent SLO, while meeting the latency SLOs for over 99% requests.

Name	Size	Latency (ms)	S1	S2	S3	S4
BERT-1.3B	2.4 GB	151	32	0	10	0
BERT-2.7B	5.4 GB	238	0	0	10	0
BERT-6.7B	13.4 GB	395	0	32	10	0
BERT-104B	208 GB	4600	0	0	0	4
MoE-1.3B	2.6 GB	150	0	0	10	0
MoE-2.4B	4.8 GB	171	0	0	10	0
MoE-5.3B	10.6 GB	234	0	0	10	0

Table 1: The first three columns list the sizes and inference latency of the models. The latency is measured for a single query with a sequence length of 2048 on a single GPU. BERT-104B’s latency is reported using a minimal degree of inter-op parallelism. The latter columns list the number of instances for each model in different model sets named as S1-S4.

## 6.1 Experiment Setup

**Cluster testbed.** We deploy AlpaServe on a cluster with 8 nodes and 64 GPUs. Each node is an AWS EC2 p3.16xlarge instance with 8 NVIDIA Tesla V100 (16GB) GPUs.

**Model setup.** Since Transformer [47] is the default backbone for large models, we choose two representative large Transformer model families: BERT [14] and GShard MoE [27] for evaluation.<sup>4</sup> In ML practice, the large model weights are usually pretrained and then finetuned into different versions for different tasks. Hence, for each model family, we select several most commonly used model sizes [5], and then create multiple model instances at each size for experimentation. Also, we design some model sets to test the serving systems under different model conditions; details about model sizes, their inference latency on testbed GPUs, and the number of model instances in each model set are provided in Tab. 1.

**Metrics.** We use *SLO attainment* as the major evaluation metric. Under a specific SLO attainment goal (say, 99%), we concern with another four measures: (1) the minimal number of devices the system needs, (2) the maximum average request rate, (3) the maximum traffic burstiness the system can support, and (4) the minimal SLO the system can handle. We are particularly interested in a SLO attainment of 99% (indicated by vertical lines in all curve plots), but will also vary each variable in (1) - (4) and observe how the SLO attainment changes.

**Simulator fidelity.** We want to study the system behavior under extensive models, workload, and resource settings, but some settings are just beyond the capacity of our testbed. Also, it is cost- and time-prohibitive to perform all experiments on the testbed for the days-long real traces. To mitigate the problem, we use the simulator introduced in §5 for the majority of our experiments, noticing that DNN model execution [19] has high predictability, even under parallel settings [27, 56].

<sup>4</sup>In this paper, we focus on non-autoregressive large models which perform inference with one forward pass, but note that the techniques proposed in this paper can be extended to auto-regressive models like GPT-3.

SLO Scale	Selective Replication		AlpaServe	
	Real System	Simulator	Real System	Simulator
0.5x	00.0%	00.0%	33.3%	33.3%
1x	00.0%	00.0%	53.5%	53.2%
1.5x	29.7%	30.2%	64.1%	64.7%
2x	36.9%	36.8%	79.0%	80.6%
3x	49.5%	48.5%	91.4%	92.1%
4x	58.6%	57.8%	96.4%	96.5%
5x	64.9%	64.0%	97.6%	97.9%
10x	83.1%	82.6%	100.0%	99.7%

Table 2: Comparison of the SLO attainment reported by the simulator and the real system under different SLO scales.

We study the fidelity of the simulator in Tab. 2. Given two model placement algorithms, we compare the SLO attainment reported by the simulator and by real runs on our testbed under different *SLO Scales*. The error is less than 2% in all cases, verifying the accuracy of our simulator. Additionally, we conduct experiments on cluster testbed for results in §6.3.

## 6.2 End-to-end Results with Real Workloads

In this section, we compare AlpaServe against baseline methods on publicly available real traces.

**Workloads.** There does not exist an open-source production ML inference trace to the best of our knowledge. Therefore, we use the following two production traces as a proxy: Microsoft Azure function trace 2019 (MAF1) [42] and 2021 (MAF2) [54]. They were originally collected from Azure serverless function invocations in two weeks, and have been repurposed for ML serving research [4, 25]. The two traces exhibit distinct traffic patterns. In MAF1, each function receives steady and dense incoming requests with gradually changing rates; in MAF2, the traffic is very *bursty* and is distributed across functions in a highly *skewed* way – some function receives orders of magnitude more requests than others. Note that most previous works [19] are evaluated on MAF1 only. Since there are more functions than models, following previous work [4, 25], given a model set from Tab. 1, we round-robin functions to models to generate traffic for each model.

**Setup.** SLO attainment depends on many factors. For each metric (1) - (4) mentioned in §6.1, we set a default value, e.g., the default SLO is set as tight as  $5 \times$  inference latency (SLO Scale=5). This forms a *default setting*, given which, we then vary one variable (while fixing others) at a time and observe how it affects the resulting SLO attainment. To change the two variables (3) and (4), which characterize traffic patterns, we follow Clockwork [19] and Inferline [8] and slice the original traces into time windows, and fit the arrivals in each time window with a Gamma Process parameterized by rate and coefficient of variance (CV). By scaling the rate and CV and resampling from the processes, we can control the rate and burstiness, respectively.

**Baselines.** We compare AlpaServe to two baseline methods:

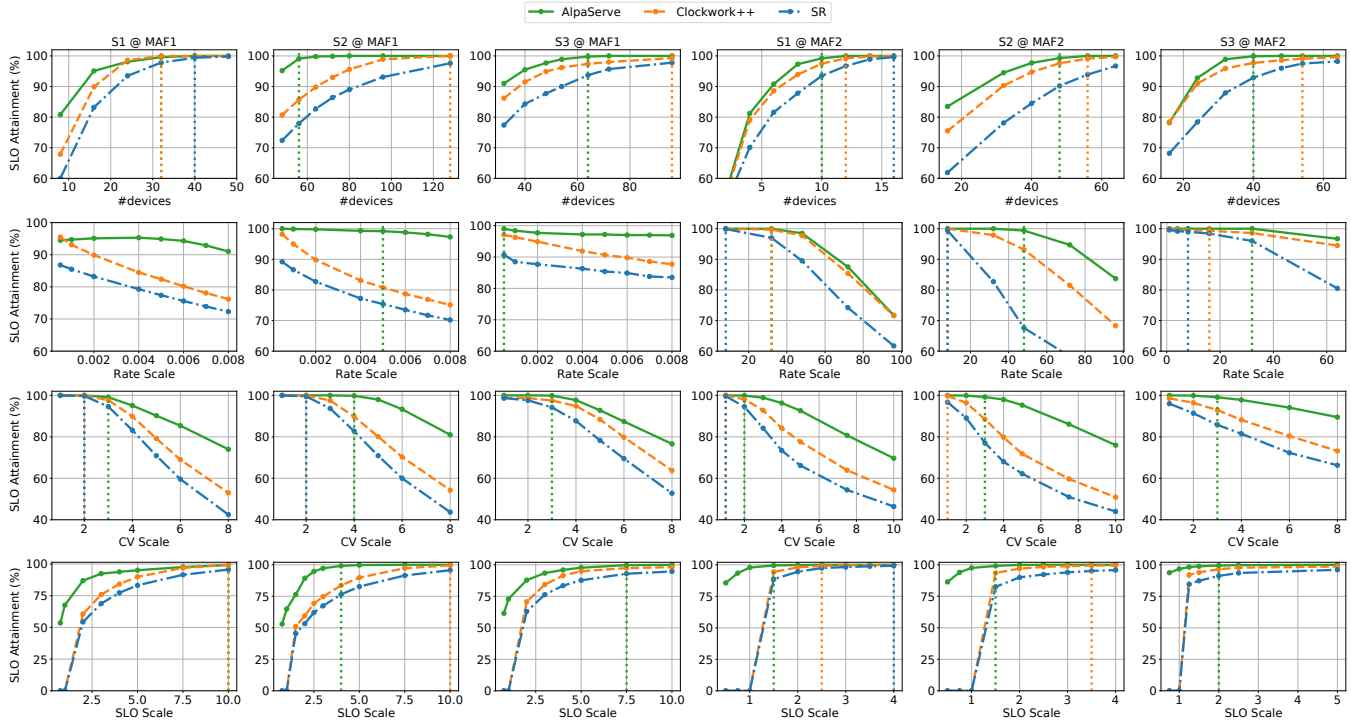


Figure 12: SLO attainment under various settings. In column S1@MAF1, we replay the MAF1 trace on the model set S1, and so on. In each row, we focus on one specific metric mentioned in §6.2 to see how its variation affects the performance of each serving system. If any, the dotted vertical line shows when the system can achieve 99% SLO attainment.

(1) *Selective Replication (SR)*: use AlpaServe’s placement algorithm without model parallelism, which mimics the policy of a wide range of existing serving systems [9, 44]; (2) *Clockwork++*: an improved version of the state-of-the-art model serving system Clockwork [19]. The original Clockwork continuously swaps models into and out of GPUs. This helps for very small models (e.g., w/ several million parameters) but incurs significant swapping overheads on larger models. For fair comparisons, we implement Clockwork++ in our simulator, which swaps models following Clockwork’s replacement strategy at the boundary of every two windows<sup>5</sup> of the trace using SR’s algorithm, but *assuming zero swapping overheads*. We believe it represents a hypothetical upper bound of Clockwork’s performance. Since all the baselines can only support models that can fit into one GPU memory,<sup>6</sup> we use model set S1, S2 and S3 from Tab. 1 in this experiment.

**SLO attainment vs. cluster size.** Fig. 12’s first row shows the SLO attainment with varying cluster sizes when serving a specific (model set, trace) pair. AlpaServe outperforms the two baselines at all times and uses far fewer devices to achieve 99% SLO attainment thanks to model parallelism. By splitting

<sup>5</sup>For MAF1, we follow Clockwork to set the window size as 60 seconds. For MAF2, we set it as 5.4K seconds.

<sup>6</sup>In our cluster testbed, the per-GPU memory is 16GB, but the actual available space for model weights is around 13GB due to the need to store activations and other runtime context.

one model replica onto  $N$  devices, AlpaServe can achieve similar throughput as if  $N$  replica were created for replication-only methods; but note AlpaServe uses only one replica of memory. Surprisingly, although we let Clockwork++ adjust to the traffic dynamically with zero overhead, AlpaServe still wins with a static placement; this is because model-parallel placement is by nature more robust to bursty traffic.

It is worth noting that replication-only methods can at most place 2 replicas of BERT-2.6B on a V100 (13GB memory budget), resulting in a substantial memory fraction, while model parallelism can avoid such memory fractions and enable more flexible placement of models.

**SLO attainment vs. rate.** Fig. 12’s 2nd row varies the rate of the workloads. For a stable trace like MAF1, AlpaServe can handle a much higher rate than baselines. While for a skewed and highly dynamic trace MAF2, whose traffic is dominated by a few models and changes rapidly, the replication-based methods have to allocate the majority of the GPUs to create many replicas for “hot” models to combat their bursty traffic; those GPUs, however, may go idle between bursts, even with frequent re-placement as in Clockwork++. In AlpaServe, each model needs fewer replicas to handle its peak traffic.

**SLO attainment vs. CV.** Fig. 12’s 3rd row varies the CV of the workloads. The traffic becomes more bursty with a higher CV, which aggravates the queuing effect of the system and increases the possibility of SLO violation. The traditional



solution to handle burstiness is by over-provision, wasting a lot of resources. AlpaServe reveals a hidden opportunity to handle this by model parallelism.

**SLO attainment vs. SLO.** Fig. 12’s 4th row shows the effect of different SLO. Previous work [19] which targets serving small models usually sets SLO to hundreds of milliseconds, even though the actual inference latency is less than 10 ms. Thanks to the intra-op parallelism, AlpaServe can maintain good performance under similar SLO when serving large models, whose inference latency can be over 100 ms. When SLO is tight, even less than the model inference time, AlpaServe favors intra-op parallelism to reduce the inference latency, which also reduces AlpaServe’s peak throughput due to the communication overhead but can make more requests to meet their SLO. When SLO becomes looser, AlpaServe will automatically switch to use more inter-op parallelism to get higher throughput.

### 6.3 Serving Very Large Models

Today’s large models may possess hundreds of billions of parameters [5, 31, 53]. To serve large models at this scale, the common practice in production is to choose the model parallelism strategy manually and use dedicated GPUs for each model [51]. To show AlpaServe has improved capability in serving very large models, we deploy model set S4 on our testbed, each requiring at least 16 GPUs to serve in terms of memory usage. As baselines, for each model, we enumerate all combinations of inter- and intra-op parallelisms on 16 GPUs. In contrast, AlpaServe searches for the optimal GPU group allocation and model placement according to the arrival traffic and tries to achieve statistical multiplexing.

**Offered load.** In the default setting, the traffic is generated via a Gamma Process with an average rate of 8 requests/s and CV of 4. We then split the requests to each model following a power law distribution with an exponent of 0.5 to simulate the real-world skewness.<sup>7</sup> Similar to §6.2, we vary one of the rate, CV, or SLO in the default setting to see how each factor contributes to the resulting performance. It is worth noting that all results presented in this section are obtained via real execution on the testbed cluster.

**SLO attainment.** Fig. 13 shows the SLO attainment of each system under various settings. Although enumerating parallelism strategies and selecting the best can improve performance, it still remains a substantial gap compared to AlpaServe. This means that the traditional way of using dedicated GPUs to serve large models is not ideal. We check the solution of AlpaServe and find it slices the cluster evenly into two groups, each with the (4, 8) inter-/intra-op parallel configuration, and groups the models in a way that balances the requests between two groups. This further proves that our motivation in §3.1 still holds for extremely large models. By

<sup>7</sup>Uniform split yielded similar results.

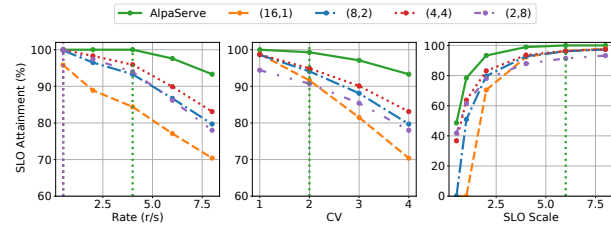


Figure 13: SLO attainment as we vary the rate, CV, and SLO scale. (8,2) means 8-way inter-op parallelism and in each pipeline stage using 2-way intra-op parallelism.

space-sharing the devices, AlpaServe can exploit new opportunities for statistical multiplexing, which is advantageous for bursty workloads but largely under-explored by prior work.

### 6.4 Robustness to Changing Traffic Patterns

Until now, AlpaServe’s good performance is based on the assumption we make in its placement algorithm that we know the arrival process in advance. In practice, the arrival process can be approximated using historical traces but the unavoidable real-world variance may make the prediction inaccurate. In this experiment, we study how AlpaServe performs if the traffic patterns change.

We reuse the same setting for S2@MAF1 in §6.2, but this time for AlpaServe and SR, we randomly slice two one-hour traces from MAF1, one is what their algorithms are assumed, while the other one is used as the actual arrival process. While for Clockwork++, we still run its algorithm directly on the actual arrival process to respect its online nature. Similarly, we vary different factors and compute the SLO attainment for each system. We repeat the experiments three times and show the average results in Fig. 14.

Unsurprisingly, SR’s performance drops significantly when traffic changes. By contrast, AlpaServe maintains good performance and still outperforms Clockwork++, an online adjustment algorithm, using a static placement generated from substantially different traffic patterns. This confirms that, in face of highly-dynamic traffic patterns, statistical multiplexing with model parallelism is a simple and better alternative than existing replication- or replacement-based algorithms.

### 6.5 Benefits of Dynamic Batching

Batching is a common optimization in other serving systems [19, 33, 34] and the choice of batch size is critical to the performance because it can increase GPU utilization and thus increase the system throughput. However, in large model scenarios, the benefit of batching is limited mainly due to two reasons. First, for large models, a small batch size will saturate the GPU, which means there is little gain to batching more requests. Second, the execution latency grows linearly with the batch size [44], so when the SLO is tight (say SLO Scale is less than 2), batching is simply not a choice.

To isolate the benefits of model parallelism and make the

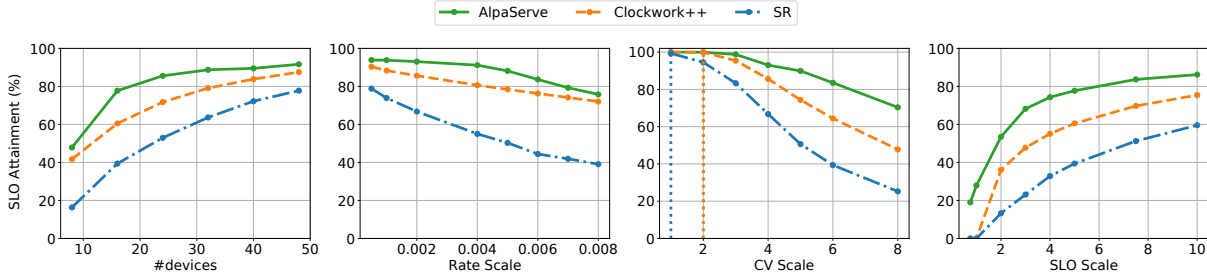


Figure 14: The actual arrival traffic for AlpaServe and SR is different from what their algorithms are assumed, while Clockwork++ runs directly on the actual traffic.

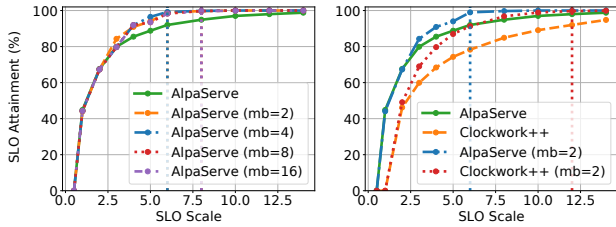


Figure 15: SLO Attainment when batching is enabled.  $mb=2$  means the maximum batch size is 2.

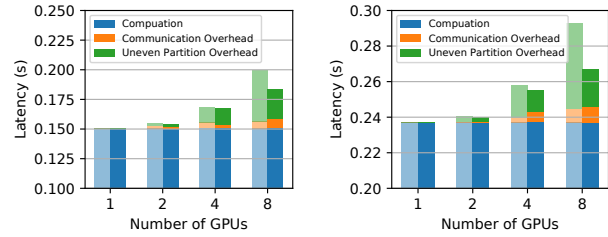
results more explainable, we decide to disable any batching in other experiments but prove that the batching strategy is purely orthogonal to the scope of this paper in this subsection. To prove this, we implement a standard batching algorithm in AlpaServe and evaluate its performance.

**Batching strategy.** When a request arrives, it will get executed immediately if any device group is available. Otherwise, it will be put into a per-model requests queue for batching. When a device group becomes idle, it will choose a model which has a replica on it and batch as many requests as possible from the requests queue of the model while satisfying the SLO requirements.

**Setup.** As the model size increases, the potential benefit of batching decreases. Therefore, we choose to evaluate model set S1. We generate a synthetic Gamma Process traffic with an average rate of 4 requests/s and a CV of 4 for each model.

**SLO attainment.** Fig. 15 (left) shows the SLO attainment achieved by AlpaServe with different maximum batch size settings under various SLO scales. When the SLO requirement is tight, any batching will violate the SLO so there is no gain with batching enabled. Also, although we choose to serve the smallest model in Tab. 1, a small batch size like 2 combined with a long sequence length of 2048 already saturates the GPU, so a larger maximum batch size brings no performance improvement. Fig. 15 (right) compares the improvement for AlpaServe and Clockwork++ with our batching algorithm enabled.<sup>8</sup> When the SLO requirement becomes loose, both AlpaServe and Clockwork++ have better SLO attainment to

<sup>8</sup>SR is left out to make the figure clearer as it is worse than Clockwork++.



(a) Transformer 1.3B. (b) Transformer 2.6B.

Figure 16: Comparison of the model parallel overhead between manual partition (lighter color) and the partition found by the automatic algorithm (darker color).

some extent. Since AlpaServe’s performance is good even without batching and batched requests with different batch sizes will incur stage imbalance and pipeline bubble in inter-op parallel, the absolute improvement of Clockwork++ is slightly better.

## 6.6 Ablation Study

In this section, we study the effectiveness of our proposed auto-parallelization (§4.1) and placement algorithms (§4.2).

**Benefits of auto-parallelization.** We show that the auto-parallelization ability allows AlpaServe to not only generalize to arbitrary model architectures but even also reduce parallelism overheads – hence improved serving performance (see §3.3 for more discussion). To see that, typical manual model-parallel parallelization strategies offered in *de facto* systems [1, 31, 32] is to assign an equal number of (transformer) layers to each pipeline stage. These strategies often fail to create balanced workloads across distributed GPUs because contemporary large models have heterogeneous layers, such as embedding operations. The extensions introduced in §4.1 automatically partition the models at the computational graph level and generate nearly-balanced stages. Empirically, as shown in Fig. 16, for 8 pipeline stages, auto-parallelization reduces the total overhead by 32.9% and 46.7% for Transformer 1.3B and 2.6B respectively, which is necessary for achieving good serving performance when model parallelism is used for serving.

**Effectiveness of the placement algorithm.** We now test the

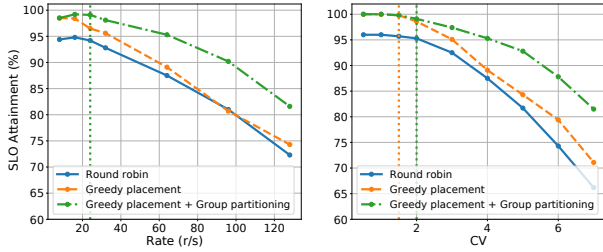


Figure 17: Ablation study of placement algorithms.

effectiveness of our placement algorithm on a synthetic workload. We serve the most challenging model set S3 (Tab. 1) on our testbed. The rate distribution of the models follows a power law distribution. The arrival pattern of each model is a Gamma process. Three variants of the placement algorithms are evaluated. *Round robin* means placing models in a round-robin fashion and using 4-stage pipelines for all groups. *Greedy placement* uses our greedy placement and 4-stage pipeline for all groups. *Greedy placement + Group partitioning* performs greedy placement plus group partitioning search. As shown in Fig. 17, both placement and group partitioning are necessary to achieve good SLO attainment. In the left subfigure, the group partitioning increases the rate by  $1.5\times$  compared to greedy placement without group partitioning over 99% SLO attainment, while round robin can never reach 99% SLO attainment. In the right subfigure, the group partitioning increases the traffic burstiness that can be handled to meet 99% SLO attainment by  $1.3\times$ .

## 7 Related Work

**Model serving systems.** There has been a proliferation of model serving systems recently. These range from general-purpose production-grade systems like TensorFlow Serving [34] and NVIDIA Triton [33], which are widely used but do not provide any support for automatic placement or latency constraints. They also include systems that are optimized for single-model serving [51] or serving of specific classes of models (e.g., transformers) [16, 51, 57]. AlphaServe targets a broader set of models and features than these systems.

For SLO-aware, distributed serving, most serving systems ignore placement-level interactions between models. Clockwork [19], for instance, primarily focuses on predictability; when scheduling, it greedily loads and executes models on available GPUs. Shepherd [52] utilizes preemption to correct sub-optimal scheduling decisions. For large models, loading model weights and preemption can easily overwhelm practical SLOs. Other systems like Clipper [9], Infaas [40], and DVABatch [10] also do not reason about the latencies of co-located models.

Nexus [44] is very related to our work in that it examines the placement of models; however, Nexus is an example of a system that takes the traditional replication approach described in §3 and, thus, misses a broad class of potential parallelization strategies that we explore in this paper.

**Inference optimizations for large models.** AlphaServe is complementary to another large body of work on optimizations for inference over large models. These include techniques like quantization [13], distillation [41], offloading [1], better operator parallelism [36], and CUDA kernel optimization [11, 26]. Some of these optimizations are intended to stem the tide of increasing model sizes; however, all of these gains are partial—the challenge of serving large models has continued to escalate rapidly despite these efforts.

**Model parallelism for training.** AlphaServe is largely orthogonal to the large body of work on model parallelism in training [23, 28, 31, 37, 56]. As described in §3, serving presents a unique set of constraints and opportunities not found in training workloads. Where these systems do intersect with AlphaServe, however, is in their methods for implementing model parallelism along various dimensions. In particular, AlphaServe builds on some of the parallelization techniques introduced in [56].

**Resource allocation and multiplexing.** The problem of how to multiplex limited resources to the incoming requests is one of the oldest topics in computer science and has been studied in different application domains [3, 29, 38]. Recent work on DL scheduling uses swapping [2], preemption [20], interleaving [55], and space-sharing [49] to realize fine-grained resource sharing. Rather, the contribution of this paper is a deep empirical analysis of the applications of these ideas to an emerging space: the serving of multiple large models.

## 8 Conclusion and Future Work

In this paper, we presented AlphaServe, a system for prediction servings of multiple large deep-learning models. The key innovation of AlphaServe is integrating model parallelism into multi-model serving. Because of the inherent overheads of model parallelism, such parallelism is traditionally applied conservatively—reserved for cases where models simply do not fit within a single GPU or execute within the required SLO. AlphaServe demonstrates that model parallelism is useful for many other scenarios, quantifies the tradeoffs, and presents techniques to automatically navigate that tradeoff space.

In the future, we will extend AlphaServe to more complicated scenarios, including serving multiple parameter-efficient adapted models (e.g., LoRA [22]), models with dependencies, and autoregressive models [5].

## 9 Acknowledgement

We thank the OSDI reviewers and our shepherd, Heming Cui, for their valuable feedback. This work is in part supported by NSF CISE Expeditions Award CCF1730628, NSFC under the grant number 62172008, and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware. Yinmin Zhong and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

## References

- [1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.
- [2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.
- [3] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-optimizing data placement for faster data-parallel computations. *Proceedings of the VLDB Endowment*, 16(4):760–771, 2022.
- [4] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [7] Copy.ai. Copy.ai: Write better marketing copy and content with ai. <https://www.copy.ai/>.
- [8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.
- [9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.
- [10] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.
- [11] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.
- [12] Robert I Davis, Ken W Tindell, and Alan Burns. Scheduling slack time in fixed priority pre-emptive systems. In *1993 Proceedings Real-Time Systems Symposium*, pages 222–231. IEEE, 1993.
- [13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 2022.
- [14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [15] Mengnan Du, Subhabrata Mukherjee, Yu Cheng, Milad Shokouhi, Xia Hu, and Ahmed Hassan Awadallah. What do compressed large language models forget? robustness challenges in model compression. *arXiv preprint arXiv:2110.08419*, 2021.
- [16] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbo Transformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [17] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
- [18] Github. Github copilot: Your ai pair programmer. <https://github.com/features/copilot>.
- [19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.
- [20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.



- [21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.
- [22] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.
- [23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- [24] Huggingface. Models - huggingface. <https://huggingface.co/models>.
- [25] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262. IEEE, 2018.
- [26] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.
- [27] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2020.
- [28] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.
- [29] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20, 2010.
- [30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [32] NVIDIA. Fastertransformer. <https://github.com/NVIDIA/FasterTransformer>.
- [33] NVIDIA. Triton inference server. <https://developer.nvidia.com/nvidia-triton-inference-server>.
- [34] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.
- [35] OpenAI. Chatgpt. <https://chat.openai.com/chat>.
- [36] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- [37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.
- [38] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.
- [39] Stewart Robinson. *Simulation: the practice of model development and use*. Bloomsbury Publishing, 2014.
- [40] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.
- [41] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

- [42] Mohammad Shahradd, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.
- [43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. MeshTensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.
- [44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.
- [45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [46] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.
- [47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [48] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.
- [49] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.
- [50] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.
- [51] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [52] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023.
- [53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
- [54] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.
- [55] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 428–440, 2022.
- [56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.
- [57] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. Pets: A unified framework for parameter-efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022.





# COCKTAILER: Analyzing and Optimizing Dynamic Control Flow in Deep Learning

Chen Zhang<sup>†£◇\*</sup> Lingxiao Ma<sup>◇</sup> Jilong Xue<sup>◇</sup> Yining Shi<sup>‡◇\*</sup> Ziming Miao<sup>◇</sup>  
Fan Yang<sup>◇</sup> Jidong Zhai<sup>†</sup> Zhi Yang<sup>‡</sup> Mao Yang<sup>◇</sup>  
<sup>†</sup>*Tsinghua University*   <sup>‡</sup>*Peking University*   <sup>◇</sup>*Microsoft Research*

## Abstract

With the growing complexity of deep neural networks (DNNs), developing DNN programs with intricate control flow logic (e.g., loops, branches, and recursion) has become increasingly essential. However, executing such DNN programs efficiently on accelerators is challenging. Current DNN frameworks typically process control flow on the CPU, while offloading the remaining computations to accelerators like GPUs. This often introduces significant synchronization overhead between CPU and the accelerator, and prevents global optimization across control flow scopes.

To address this challenge, we propose COCKTAILER, a new DNN compiler that co-optimizes the execution of control flow and data flow on hardware accelerators. COCKTAILER provides the *uTask* abstraction to unify the representation of DNN models, including both control flow and data flow. This allows COCKTAILER to expose a holistic scheduling space for rescheduling control flow to the lower-level hardware parallelism of accelerators. COCKTAILER uses a heuristic policy to find efficient schedules and is able to automatically move control flow into kernels of accelerators, enabling optimization across control flow boundaries. Evaluations demonstrate that COCKTAILER can accelerate DNN models with control flow by up to 8.2× over the fastest one of the state-of-the-art DNN frameworks and compilers.

## 1 Introduction

In deep neural networks (DNNs), control flow plays a crucial role in accomplishing sophisticated tasks, akin to its usage in general programming languages. Examples of this include iterating over sequential data like text and time steps, activating different components of the model based on input-data-driven conditions, dynamically skipping some computation based on runtime decisions for efficient computation, and recursively

traversing tree-based data structures. A DNN program is typically divided into two parts: control flow and data flow. The data flow is typically represented as a graph of DNN operators, which can be efficiently executed on specialized accelerators such as GPUs. The control flow, on the other hand, is either implemented as a special operator [4] or by directly reusing the built-in statements of programming languages [36], and is typically executed on a CPU. Therefore, the control flow and data flow are executed alternatively in an entire DNN computation: the control flow determines which part of the data flow should be executed, and then the corresponding data flow is sent to accelerators for processing and the result is obtained, which is used to decide the next step of control flow.

However, the interleaved execution paradigm on both sides in existing DNN frameworks often introduces significant efficiency issues. First, the control flow and data flow require frequent synchronization between the CPU and accelerator (e.g., for checking conditions based on results), resulting in significant communication overhead (e.g., across PCIE) in the critical path. Second, the control flow in a DNN program often establishes explicit boundaries between data flow operators, which prevents their holistic optimization for maximum efficiency, such as fusing two operators across a loop scope. Lastly, the control flow implicitly serializes the execution of data flow operators that could potentially be executed in parallel. We have observed that these overheads are prevalent in existing DNN models and can often occupy as much as 72% of the total time in PyTorch. These efficiency issues not only introduce obstacles to dynamic model developments but also make many optimizations, e.g., dynamically skipping some computation, hard to achieve theoretical speedup.

Based on our observation of DNN workload patterns, the fundamental reason for the inefficiency is the *parallelism mismatch* between the control flow and data flow. In particular, control flow operations, such as loops, branches, and recursion, are single-thread semantic and execute in a strictly sequential order. However, the data flow operators are parallelizable, running on multiple parallel threads (e.g., GPU cores) and synchronizing periodically across different scopes

<sup>£</sup>Tsinghua University, BNRist

<sup>\*</sup>Work is done during the internship at Microsoft Research.



of threads (between operators or thread blocks). To control the execution flow of a parallel program, the mismatch between control flow and data flow forces existing practices to place the control flow either outside the DNN operators (e.g., in existing DNN frameworks) or inside an individual DNN operator, through implementing custom kernels in an ad-hoc way (e.g., Relu operator). This can be either inefficient or unscalable to support the increasing demands for control flow in DNN workloads.

In this paper, we present a new DNN compiler, COCKTAILER, that addresses the challenges of co-optimizing control flow and data flow in a single space. COCKTAILER is based on three key insights observed from studying DNN models and modern accelerators. First, the data flow in DNNs is inherently a multi-level parallel program, where individual operators are executed in different hardware parallelism, such as threads, thread blocks, or kernels in GPUs. Second, the control flow operations in DNNs are mostly applied at the operator level, where all lower-level parallelisms share the same control result. This implies that the control flow can be rescheduled to the low-level parallelism by replicating the control logic to all parallel tasks at different levels. Most importantly, modern hardware accelerators, such as GPUs, are designed to support the control logic in their low-level programming languages in each thread, which makes this rescheduling approach feasible in practice.

Based on these insights, COCKTAILER introduces the *uTask* abstraction as the primitive execution unit of a DNN program for both control flow and data flow. An operator in data flow can be naturally decomposed into different levels of granularity of *uTasks* aligned with its computation parallelism. For control flow, COCKTAILER introduces three types of special *uTasks*: loop *uTask*, branch *uTask*, and *uTask* reference, to represent the program with control flow as a special *uTask*. By unifying the DNN program into the *uTask* granularity, COCKTAILER creates a holistic space for co-scheduling control flow *uTasks* with compute *uTasks*, i.e., by assigning the control flow *uTask* to the most efficient parallel level with data dependencies resolved correctly. To facilitate this scheduling, COCKTAILER proposes a scheduling mechanism and a traverse-based bottom-up scheduling policy that incorporates all control flow optimizations such as function inline, loop unroll, and recursion unroll.

As a result, COCKTAILER is able to automatically move control flow operations, such as loops or branches, into accelerator side when applicable, enabling more optimization opportunities across the control flow boundary. COCKTAILER is built on top of general DNN tensor compilers by leveraging their kernel generation capabilities for *uTask*, allowing it to adapt to different accelerators such as CUDA GPUs and ROCm GPUs easily. COCKTAILER’s approach can be applied to both DNN frameworks that implement control flow as special operators or language-built-in statements, by only compiling the sub-programs that can be optimized by COCK-

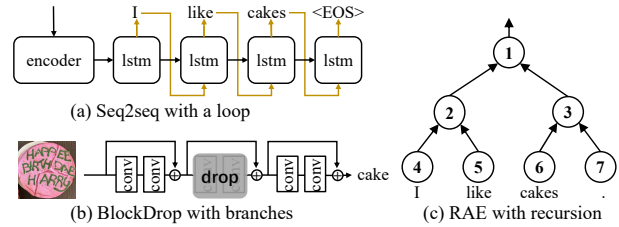


Figure 1: Models with control flow

TAILER. Evaluation with 7 typical DNN models on CUDA GPUs and ROCm GPUs shows that COCKTAILER accelerates these models by up to  $8.2\times$  over the fastest one of state-of-the-art DNN frameworks and compilers. Furthermore, the evaluation shows that COCKTAILER not only reduces the overheads introduced by control flow but also enables scenarios like dynamically skipping some computation by achieving real speedup. The code has been open-sourced<sup>1</sup>.

## 2 Background and Motivation

DNNs have been successfully applied in many areas, such as computer vision, speech, and natural language. Meanwhile, the concept of control flow in programming languages is introduced to deep learning. The architecture of DNN models rapidly evolves from sequential feed-forward layers [15, 23, 38] to structures with complex control logic [16, 39–41, 46, 47], enabling dynamic computation and adaptability within the network architecture:

- **Dynamic computation.** Control flow enables constructing dynamic computation architectures, which can adapt their structure during runtime. For example, the loops are widely used to handle variable-length sequences (e.g., text, speech, time-series data) in RNNs [16, 40, 58] and Transformers [43].
- **Conditional computation.** Control flow enables the execution of specific parts of the model based on certain conditions [7, 24] like executing different parts of the model for images with different resolution.
- **Efficient computation.** Control flow can help reduce the computational resources required by DNN models by selectively executing parts of the model based on input data or intermediate results, e.g., the early-exiting mechanism [46, 47] that can skip some layers on easy input samples. Besides, control flow can be leveraged to adapt DNN models to different environments (e.g., different hardware accelerators) by trading off computation cost and model performance via control flow [26].

Dynamic computation for structural data is a common requirement in modern deep learning models. For instance, nearly 27% of the 52 models in PyTorch Hub (as of commit ID 1c747e2) contain structural data (e.g., sequence, tree). Furthermore, a survey on dynamic DNN models [13] indicates

<sup>1</sup>[https://github.com/microsoft/nnfusion/tree/cocktailer\\_artifact/artifacts](https://github.com/microsoft/nnfusion/tree/cocktailer_artifact/artifacts)

that conditional computation and efficient computation are promising research directions.

In programming languages, control flow constructs are typically categorized into the following types: sequence, branch, loop, and subroutine (function). Similarly, a majority of DNN models with complex control flow can be categorized into models with loops for temporal-wise dynamism (e.g., LSTM [16], NASRNN [58], Seq2seq [40]), models with branches to skip computation (e.g., BlockDrop [47], SkipNet [46]), and models with tree-based architecture via recursion (e.g., RAE [39], Tree-LSTM [41]). We discuss the three representative categories with their representative models, as shown in Figure 1.

- **Loop.** Seq2seq [40] can generate arbitrary-length sequences. It contains a while loop that continues to emit new tokens until an end-of-sequence (EOS) token.
- **Branch.** BlockDrop [47] is a convolution neural network that can drop some convolution layers. Each layer is implemented by an if statement with two branches for whether executing the branch or not.
- **Recursion.** RAE [39] computes the embedding of a parse tree by traversing the tree. It can be implemented by a depth-first search using recursion.

To support these emerging DNN models, there are two mainstream approaches. The first one, represented by TensorFlow of version 1.x [4], supports these complex model architectures by introducing a set of control flow operators [52] like `Enter` and `NextIteration`. Then, these control flow operators are executed in the framework runtime with the CPU threads. The second one, represented by PyTorch [36] and JAX [11], leverages the programming language to represent and execute the control flow. For example, in PyTorch, algorithm designers program control logic in Python, and these control flow statements are running in the Python runtime.

Both approaches schedule data flow operators onto the accelerators while maintaining control flow in the CPU side for execution. The reason is the *parallelism mismatch* between the control flow and the data flow. Specifically, different from data flow operators (e.g., matrix multiplication) that have internal data parallelism, control flow operations are represented as single-thread computation. Modern hardware accelerators have massive hierarchical parallel processing units. For example, GPUs contain many parallel streaming multiprocessors (SMs) and each SM has many parallel cores. This architecture aligns with data flow operators' parallelism, but it is hard to schedule the single-thread control flow to the massive hierarchical parallel processing units for execution. Therefore, current practices [4, 11, 36, 52] schedule control flow operations to the CPU side for execution. Such approaches introduce boundaries between DNN operators of different basic blocks, resulting in performance issues.

Figure 2 compares JAX's performance of executing the three models via dynamic control flow to executing the corresponding traced static graph that has removed all the control

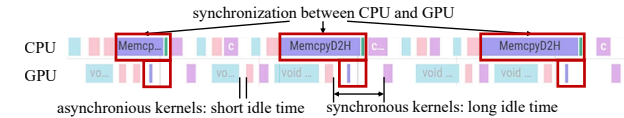
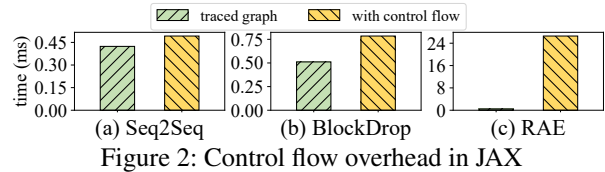


Figure 3: Timeline of BlockDrop in JAX. The data copy from GPU to CPU (MemcpyD2H) causes synchronization.

flow computation and only remains data flow computation. Compared with the traced graph baseline, the control flow computation of JAX causes  $1.16\times$ ,  $1.54\times$ , and  $56.22\times$  slow-down in Seq2Seq, BlockDrop, and RAE, respectively. More results can be found in §5. The loop in Seq2seq and branches in BlockDrop use control flow operators. The recursion in RAE is executed in Python. Both approaches cannot match the performance of static traced graphs.

The performance issue comes from the following parts.

**(1) Boundary overheads.** Executing data flow operators on the accelerator and control flow operations on the CPU can incur synchronizations between the CPU and the accelerator. Take the BlockDrop model on a CPU-GPU system as an example, the DNN operators in the branch body are executed in the GPU side, while the branch operation is executed in the CPU side. The CPU stalls when waiting for the GPU to provide the data required for deciding the branch target, and then the GPU stalls to wait for the CPU to check the branch condition and send the following operations to the GPU. The boundary overheads mainly contain the communication between the CPU and the accelerator and the kernel launching. This boundary may also break the asynchronous execution in the accelerator side.

Figure 3 shows part of the timeline of JAX executing the BlockDrop model that the CPU-GPU synchronization not only has high synchronization overheads but also breaks the asynchronous execution and causes a long idle time without computation in the GPU side.

**(2) Boundary limits the optimization scope.** Executing control flow and data flow on separate sides divides the DNN program into sub-programs, each representing a static data flow that can be executed on the accelerator side. Many DNN optimizations (e.g., Rammer [28], kernel fusion [35, 56], etc.) are limited to only optimizing these sub-programs, resulting in sub-optimal performance. Consider a multi-layer LSTM model as an example: the DNN operators in LSTM cells across different layers can be scheduled for parallel execution. However, the loop control flow constrains the DNN optimizations within a cell, resulting in overlooking this parallelism.

**(3) Boundary prevents parallelism in DNN programs.** This boundary makes the DNN programs in different con-

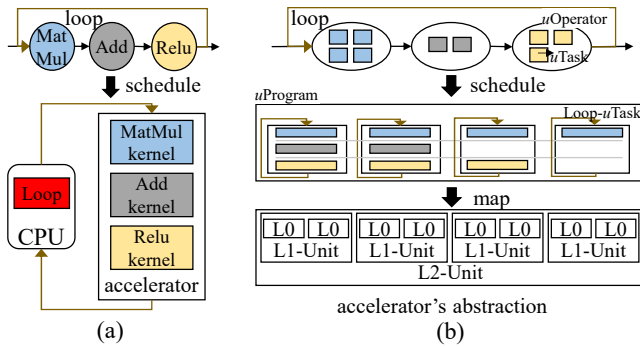


Figure 4: System overview of DNN computation in (a) existing DNN frameworks (e.g., JAX), and (b) COCKTAILER

control flow statements executed sequentially due to the synchronizations between the CPU and the accelerator, which may prevent possible inter-operator parallelism. Take the RAE model as an example, the recursion builds up a tree-based architecture where operators without dependencies can be executed concurrently. However, because the control flow can only be executed sequentially, operators of nodes without dependencies are executed sequentially.

**Observations and opportunities.** Given the fundamental limitations of current approaches described above, it is desirable to schedule the DNN programs including control flow and data flow in a single space (i.e., the accelerator side). However, it is challenging to achieve this because of the parallelism mismatch between control flow and data flow. Fortunately, control flow in DNN programs is applied across DNN operators, that is to say, the DNN operators’ computation under control flow shares the same control logic. On the other hand, most hardware accelerators (e.g., GPUs) have the ability to execute control flow instructions. If we represent control flow in a finer granularity that can be properly mapped to the parallel processing units for execution, we can schedule both data flow and control flow to the accelerator side.

### 3 COCKTAILER Design

The observation in §2 motivates COCKTAILER, a DNN compiler for co-optimizing control flow and data flow in a single space. Figure 4 shows the overview of COCKTAILER. First, COCKTAILER takes a DNN program with control flow and data flow as input, where each operator in the data flow is a *uOperator* that consists of independent and homogeneous *uTasks*. Each *uTask* can be scheduled to one compute unit of the accelerator. Second, instead of scheduling control flow on the CPU side and data flow on the accelerator side separately, COCKTAILER schedules control flow and data flow inside the program in a single space. COCKTAILER will generate the *uProgram* representation for the program, which contains multiple independent *uTasks* (e.g., the Loop-*uTasks* in Figure 4(b)) that can be scheduled to the parallel compute units in hardware accelerators for execution. Each *uTask* represents

both the control flow and data flow logic of one compute unit.

COCKTAILER abstracts an accelerator of massive parallelism as multiple levels of parallel processing units. In each level, there are parallel and homogeneous processing units, which construct a higher level of processing unit. This hardware abstraction naturally aligns with common hardware accelerators. Take the NVIDIA GPU as an example, there are multiple homogeneous streaming multiprocessors (SMs) in a GPU, where each SM consists of multiple homogeneous cores. Therefore, NVIDIA GPUs can be mapped as an architecture with 3 levels of parallel processing units in COCKTAILER’s hardware abstraction shown as Figure 4: L0 is the core (thread); L1 is the SM (thread block); L2 is the GPU device (kernel).

The example loop structure in Figure 4 is scheduled as a *uProgram* mapped on the 3-level accelerator. The *uProgram* consists of 4 loop-*uTasks* for 4 L1-Units respectively and each loop-*uTask* is mapped to a L1-Unit for execution. Both the data flow operators and the loop are scheduled into the loop-*uTasks*. Take the first loop-*uTask* as an example, it has a loop control flow and a list of *uTasks* for data flow operations containing 1 MatMul *uTask*, 1 Add *uTask*, and 1 ReLU *uTask*.

The concepts of *uTask*, *uOperator*, and *uProgram* are described in detail in §3.1. And the *uProgram* scheduling is illustrated in §3.2.

#### 3.1 *uTask*-based DNN Program

To co-schedule the control flow and the data flow of a DNN program to accelerators with massive parallel units, COCKTAILER defines the DNN program in fine grained with the concept of *unit-task* (*uTask*). Specifically, *uTask* is defined as the computation logic that can be scheduled to one of the multi-level processing units in hardware accelerators for execution. Note that the computation in a *uTask* can be a list of other *uTasks*, i.e., a nested *uTask*. *uProgram* represents the execution plan of the *uTask*-represented DNN program mapped to a level of parallel processing units on the hardware.

***uTask* and *uOperator* for data flow operators.** As Figure 5(a)(c) show, a data flow operator is represented as a group of independent and homogeneous *uTasks* where each *uTask* is the computation to be scheduled to one processing unit. Specifically, each *uTask* takes a slice of the input tensor via `get_input_data()` and executes the corresponding computation defined in `compute()`. Then, a *uOperator* is defined as the collection of all *uTasks* of the corresponding data flow operator. The *uTasks* of a *uOperator* are indexed by the logical `uTask_id` and called by `compute(uTask_id)`. The total *uTask* count in an *uOperator* is reported by `get_uTask_num()`. When all *uTasks* in an operator are executed, the execution of this operator is finished.

Data flow operators (e.g., matrix multiplication) are usually implemented as multiple independent and homogeneous tasks that are scheduled to the massive parallel units of accelerators



<pre>interface uTask {     void compute();     void get_input_data(); };</pre>	<pre>interface NestedUTask: uTask {     void compute();     void get_input_data();      vector&lt;uTask&gt; body_uTasks; };</pre>	<pre>interface uOperator {     void compute(uTask_id);     size_t get_uTask_num();      set&lt;uTask&gt; uTasks; };</pre>	<pre>interface uProgram {     void compute(uTask_id);     size_t get_uTask_num();     set&lt;uTask&gt; uTasks;     size_t unit_level; };</pre>
(a)	(b)	(c)	(d)

Figure 5: The definition of *uTask*, *uOperator* and *uProgram*

for execution. Each task consumes a slice of the input tensor, processes the corresponding computation over the input slice, and produces a slice of the output tensor. Take the NVIDIA GPU as an example, the kernel of an operator (e.g., matrix multiplication) is scheduled as multiple thread blocks and each of them is mapped to an SM for processing a tensor slice. Furthermore, a thread block is scheduled as multiple threads and mapped to cores for processing a tensor slice. Therefore, the concept of *uTask* is not only natural to represent the fine-grained computation of data flow operators, but also aligns with the hardware architecture of multi-level parallel processing units in accelerators.

***uTask* for control flow.** It is natural to represent data flow operators with *uTasks* due to the internal data parallelism that can be divided into parallel tasks. However, different from DNN operators, the control flow cannot be divided into such parallel tasks. To enable the scheduling of control flow on the parallel processing units, we need to bridge this gap of the mismatching between the control flow computation and the massive parallelism in the accelerator.

Control flow operation applies to a scope of DNN operators in DNN programs. When the DNN operators can be divided into independent and homogeneous *uTasks*, controlling the DNN operators is equal to applying control flow computation on each *uTask*. For example, assuming there is a loop structure that has a matrix multiplication operator in the loop body, compared with executing the loop over the operator, it is equally that let each unit of the hardware accelerator process the loop control flow over the *uTask* of the operator. If we apply such control flow on the scope of the fine-grained representation of these DNN operators, we can schedule such computation including the control flow to the parallel processing units of the hardware accelerators. That is to say, we can represent control flow in the *uTask* granularity by replicating the control flow computation to the multi-level parallel units that each unit executes the control flow independently and controls the *uTasks* scheduled on the unit.

According to the observation, COCKTAILER represents control flow operations as *NestedUTasks* defined in Figure 5(b), where the computation in the body is represented in the *body\_uTasks*. These *uTasks* have data dependencies and should be executed sequentially on one processing unit. Different from data flow operators that the *get\_input\_data()* extracts a slice of the input tensor, the input data of the *uTasks* in the *body\_uTasks* of control flow is related to the results of the control flow. For example, in the LSTM model, the

```
1 interface LoopUTask: NestedUTask {
2     void compute();
3     void get_input_data();
4     void control_flow();
5     vector<uTask> body_uTasks;
6 };
```

(a) Loop-*uTask*

```
1 interface BranchUTask: NestedUTask {
2     void compute();
3     void get_input_data();
4     void control_flow();
5     vector<uTask> then_uTasks;
6     vector<uTask> else_uTasks;
7 };
```

(b) Branch-*uTask*

Figure 6: Control flow *uTasks*

*uTasks* in the body of the loop control flow require different values of the loop counter in different loop steps. Therefore, the *get\_input\_data()* for control flow should prepare the input data with consideration of the results of control flow. Note that different control flow operations have different data access patterns in the *body\_uTasks*. We will discuss it in detail in the following.

According to Section 2, there are three types of control flow in DNN programs: loop, branch, and recursion. Therefore, COCKTAILER defines the concepts of loop-*uTask*, branch-*uTask*, and *uTask* reference correspondingly to represent the fine-grained *uTask* for control flow in DNN programs.

(1) *Loop-uTask*. Figure 6(a) shows the *uTask* definition for the loop control flow. COCKTAILER currently supports two types of loop control flow, i.e., for loop and while loop. The *control\_flow()* interface implements the for loop or the while loop condition. The body computation of a loop represented in *uTasks* is implemented in *body\_uTasks*. Note that the *body\_uTasks* is executed multiple times in a loop with different input data in each loop step. For example, in the LSTM model, the computation of a LSTM cell in each loop step requires the same model parameter tensors but different loop counter tensors and state tensors. The *get\_input\_data()* interface needs to prepare the corresponding tensors in each loop step.

(2) *Branch-uTask*. Figure 6(b) shows the *uTask* definition for the branch control flow. The *control\_flow()* interface implements the condition computation in the branch. The branch-*uTask* has *then\_uTasks* and *else\_uTasks* to indicate the computation of two branches represented in *uTasks*, respectively. The *get\_input\_data()* interface returns the required data for a branch indexed by the condition result.

(3) *Function*. A function can be natively represented



```

1 ScheduleOperator(op, D, unit_level, config);
2 ScheduleControlFlow(g, D, unit_level, config);
3 Config SetResource(D, unit_level, resource);

```

Figure 7: Scheduling interfaces

with `NestedUTasks` that each `uTask` represents the computation in the function body in the fine-grained `uTasks` in the `body_uTasks`. The `get_input_data()` interface prepares input data tensors and The `compute()` interface executes the `uTasks` in `body_uTasks` sequentially.

(4) *Recursion and uTask reference.* Functions can be represented with `uTasks`. However, recursion is a special case in functions that a function may call itself in the function body. That is to say, a `uTask` may have itself in its `body_uTasks`. To support recursion, COCKTAILER introduces `uTask` reference to reference a `uTask` definition. The `uTask` reference can be considered as a function call to a `uTask`. The difference between a reference and a `uTask` is that the reference is a declaration for a `uTask` while a `uTask` defines the computation in a function. When executing a reference, COCKTAILER will find its `uTask` definition and execute this `uTask`.

The `uTask` abstraction in COCKTAILER is a general abstraction to represent control flow. We show how to represent loop, branch, function and recursion with `uTask` as most of current DNN models only contain these structures. More types of control flow can be represented by inheriting the `NestedUTask`.

**uProgram** The generated execution plan of the whole input DNN program is represented by a `uProgram`. The `uProgram` contains independent `uTasks`, each of which is the compute logic scheduled to one processing unit of the `unit_level` of the accelerator. The `uTasks` can be executed by `compute`, and the total `uTask` count of the `uProgram` is reported by `get_utask_num`.

The `uTask` abstraction enables COCKTAILER to represent DNN programs with data flow operators and control flow in a fine granularity for accelerators with massive parallelism. This representation opens a new space for co-scheduling control flow and data flow.

### 3.2 uProgram Scheduling

The `uTask` representation for DNN programs opens a large scheduling space for co-optimizing control flow and data flow in a single space. Instead of the pre-defined schedule in existing frameworks that executes data flow on the accelerator side while executes control flow on the CPU side, COCKTAILER chooses to explore this scheduling space at compile-time. To achieve this, COCKTAILER separates the scheduling policy from its mechanism. On the mechanism side, COCKTAILER provides *scheduling interfaces* with *scheduling constraints*. On the policy side, COCKTAILER provides a *traverse-based scheduling policy*. Note that the scheduling is generally designed for operators of `uTask` representation and can be executed automatically.

**Scheduling interfaces.** COCKTAILER provides three interfaces `ScheduleOperator`, `ScheduleControlFlow` and `SetResource`, to facilitate the scheduling process, as shown in Figure 7. Specifically, `ScheduleOperator` schedules an operator `op`, which can be either a data flow `uOperator` or a solely-scheduled control flow operation, into the target `uProgram` with `unit_level` of the accelerator `D`. The `config` describes the current scheduling status including the target `uProgram` and is initialized by the `SetResource`. `ScheduleOperator` will set the target `uProgram` to `NULL` if it fails to schedule the `uOperator`. Similarly, `ScheduleControlFlow` schedules a control flow operation whose body has been scheduled to the required `unit_level` under the scheduling `config`, and returns `NULL` when failing to schedule this control flow. To ensure correctness, both schedule functions will add necessary `barriers` to enforce the desired `uTask` dependency. Moreover, as control flow should control the `uTasks` in the body, a scheduling constraint is shown below.

**Constraint 1** *The unit\_level of control flow should not be lower than the unit\_level of data flow in the body.*

COCKTAILER also has a profiler that measures the execution time for a `uProgram`. The profiled information could guide a policy on deciding whether to schedule a `uProgram` to the `unit_level` of the accelerator.

**Traverse-based bottom-up scheduling policy.** Algorithm 1 describes a traverse-based scheduling policy to show how to use the interfaces and the profiler to schedule control flow and data flow in a single space to the accelerator side. This policy takes a DNN program `g` represented as control flow operations and `uOperators` in data flow and the accelerator `D` as input and returns a list of scheduled `uPrograms` on this accelerator. The policy also accepts a `unit_level` parameter indicating the highest scheduled `unit_level` of the operators inside the graph `g` or `NULL` if the operators inside the graph are not scheduled yet, which is the initial case. If the input program has multiple operators, COCKTAILER will put these operators into a function operator before scheduling.

Initially, this policy schedules all the data flow `uOperators` to `uProgram` (line 4 and line 21-27). The policy continues by progressively trying to schedule more parts of the program to the same `uProgram` if the profiler suggests this schedule could reduce the overall execution time (line 5-27). Specifically, the policy will recursively traverse the program (line 7) until it only contains a `uOperator` (line 3-4) and schedule it to the `uProgram` via `ScheduleProgram` which achieves this via `ScheduleOperator`. During the traverse, if all the operations in the input program are scheduled to accelerator's units (line 21), the policy will try to schedule this program (i.e., the control flow) to the `uProgram` (line 21-27) via `ScheduleProgram`. `ScheduleProgram` implements scheduling an input program `g` to a `unit_level` of the accelerator `D`. Note that the input `g` is either a graph of operators

---

**Algorithm 1: Traverse-based Scheduling Policy**


---

**Data:**  $g$ : DNN program represented with  $uOperator$ ;  $D$ : accelerator  
**Result:**  $uProgram$

```

1 Function Schedule( $g, D, unit\_level = NULL$ ):
2    $ulevel = unit\_level, ulevel_{max} = D.unit\_levels.size() - 1, uProgs = []$ ;
3   if  $g \in D.Operators$  and  $ulevel$  is  $NULL$  then
4      $ulevel = 0$ ;
5   if  $ulevel$  is  $NULL$  then
6     for  $op \in g.TopoSort()$  do
7        $g_{op}, ulevel_{op} = Schedule(op, D, NULL)$ ;
8        $ulevel = \max(ulevel, ulevel_{op})$ ;
9        $uProgram_p = uProgs[-1]$ ;
10       $ulevel_m = \max(ulevel_{op}, uProgram_p.ulevel)$ ;
11      if  $ulevel_m < ulevel_{max}$  then
12         $g_{merge} = uProgram_p.g + g_{op}$ ;
13         $g_{merge}, ulevel_{merge} = Schedule(g_{merge}, D, ulevel_m)$ ;
14        if  $ulevel_{merge} < ulevel_{max}$  then
15           $uProgs[-1] = g_{merge}.uProgs[0]$ ;
16           $ulevel = \max(ulevel, ulevel_{merge})$ ;
17          continue;
18         $uProgs.append(g_{op}.uProgs)$ ;
19    else
20       $uProgs = g.uProgs$ 
21    if  $ulevel < ulevel_{max}$  then
22      for  $ulevel_{cur} \in range(ulevel, ulevel_{max})$  do
23         $uProgram_{cur} = ScheduleProgram(g, D, ulevel_{cur})$ ;
24        if  $uProgram_{cur}$  is not  $NULL$  then
25          if  $Latency(uProgram_{cur}) < Latency(uProgs)$  then
26             $uProgs = [uProgram_{cur}]$ ;
27             $ulevel = ulevel_{cur}$ ;
28       $g.uProgs = uProgs$ ;
29      return  $g, ulevel$ ;
30 Function ScheduleProgram( $g, D, unit\_level$ ):
31   //  $g$  is a graph of operators in  $uTask$  representation or a control
32   // flow operation that the body has been scheduled
33    $resource = GetResource(D, unit\_level)$ ; // calculate resource
34    $cfg = SetResource(D, unit\_level, resource)$ ;
35   if  $g \in D.ControlFlow$  then
36     return ScheduleControlFlow( $g, D, unit\_level, cfg$ );
37   else
38     for  $op \in g.TopoSort()$  do
39       ScheduleOperator( $op, D, unit\_level, cfg$ );
40     return  $cfg.uProg$ ;

```

---

in  $uTask$  representation (including  $uOperators$  and scheduled control flow) or a control flow whose body has been scheduled as  $uProgram$ . Therefore, *ScheduleProgram* calls *SetResource* to configure the scheduling and leverages the config to schedule the program with *ScheduleOperator* and *ScheduleControlFlow*. The  $unit\_level$  is maintained as Constraint 1 during scheduling.

Several optimizations can be employed to reduce the scheduling time. For conciseness, these optimizations are not explicitly shown in the pseudo code. For example, trials on different  $unit\_levels$  (line 22) can be performed in parallel.

**Scheduling optimizations** There are three optimization opportunities during the scheduling, depending on the inputs and the DNN programs.

*Function inline.* To remove function call overhead, COCKTAILER converts a function control flow without recursion to

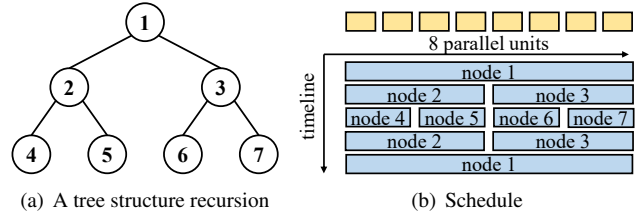


Figure 8: Parallel execution of recursive calls

a sequence of computation. It removes the function control flow boundary and applies DNN optimizations to a larger program scope.

*Loop unroll.* COCKTAILER unrolls the loop control flow with some steps to explore more optimization opportunities. For example, unrolling the loops in a multi-layer RNN model can expose parallelism between RNN cells. Loop unrolling is applied during scheduling and is evaluated to decide whether to enable this unroll.

*Recursion unroll.* It is similar to loop unroll in that the recursion is also able to be unrolled to explicitly expose the recursion tree structure. COCKTAILER applies this optimization to DNN programs to unroll the recursion structure several times to expose more optimization opportunities. For example, the unrolled recursion tree can naturally expose parallelism between recursive calls, which can be leveraged for concurrent execution. Figure 8 shows an example of recursion unroll. By unrolling the recursive calls, computation without dependencies (e.g., nodes 2 and 3 in Figure 8) can be executed concurrently. Recursion unroll is applied during scheduling. The scheduler will evaluate the unrolled results to decide whether to enable this unroll and schedule the unrolled body to different computation units.

These optimizations are in *ScheduleControlFlow*. The scheduler will try to enable these optimizations and evaluate the performance with some sample data to decide whether to enable optimizations or not.

## 4 Implementation

COCKTAILER is implemented by about 10000 lines of code including Python and C++ on top of PyTorch [36] and Rammer [28]. COCKTAILER does not require any effort from model developers, who can still work on a native PyTorch program. COCKTAILER first exports the PyTorch program to an ONNX graph with built-in loop and branch operators and an extended invoke operator for representing recursion. With the converted ONNX graph, COCKTAILER automatically performs the scheduling of data flow and control flow, and applies control-flow-related optimizations described in §3. Then, COCKTAILER wraps the generated code as a customized PyTorch operator and replaces the PyTorch program with a call to this operator.

We implemented COCKTAILER for NVIDIA GPUs and AMD GPUs because they are the most popular accelerators

```

1 // y = matmul(x, w); out = tanh(y);
2 __device__ void UProg<NumUTask=2>(float* x,
  float* w => float* out | char* tmp,int id) {
3   if (id == 0) {
4     float *y = (float*)tmp;
5     MatmulUOp.compute(x, w => y, id=0);
6     MatmulUOp.compute(x, w => y, id=2);
7     Barrier(blocks={0,1});
8     TanhUOp.compute(y => out, id=0);
9   } else if (id == 1) {
10    float *y = (float*)tmp;
11    MatmulUOp.compute(x, w => y, id=1);
12    MatmulUOp.compute(x, w => y, id=3);
13    Barrier(blocks={0,1});
14    TanhUOp.compute(y => out, id=1);
15  }
16 }

```

Figure 9: Example of *uTask*

for DNNs. In the rest of this section, we describe the details about implementing COCKTAILER for NVIDIA CUDA GPUs, and briefly describe our implementation on AMD GPUs. COCKTAILER can be ported to other accelerators if they align with the hardware abstraction described in §3 and expose APIs to control the units (e.g., Graphcore IPU).

## 4.1 COCKTAILER on NVIDIA CUDA GPUs

As described in §3, an NVIDIA GPU can be abstracted as a 3-level hardware. COCKTAILER implements the ScheduleOperator interface on top of Rammer [28], AutoTVM [6], Ansor [53], Roller [57], and manually-implemented kernels. Specifically, COCKTAILER first obtains the source code of each dataflow operator on the given unit\_level by choosing from existing manual implementations of simple operators like element-wise ones or by tuning the operator with AutoTVM, Ansor, or Roller. COCKTAILER then leverages Rammer to convert the data flow operators' kernel source code to a *uOperator* with multiple *uTasks*. After that, COCKTAILER schedules the program and generates the kernel code for the control flow body.

### 4.1.1 Code Generation for Nested-*uTask*

**Overall structure** A list of *uOperators* inside a function will be scheduled to a *uProgram* with multiple Nested-*uTasks*. It will be converted to a function with pointers to the related tensors. Specifically, we use (A => B | C) to represent a function with tensor A as input, tensor B as output, and tensor C as a buffer saving intermediate results. The function also accepts a *uTask\_id* parameter for indexing the *uTasks* in the *uProgram*. Figure 9 provides an example Function-*uProgram* with a *matmul* *uOperator* implemented by 4 *uTasks* and a *tanh* *uOperator* implemented by 2 *uTasks*. This Function-*uProgram* contains 2 *uTasks*, each of which contains 2 *matmul* *uTasks* and 1 *tanh* *uTasks* in the *body\_uTasks* with proper barrier inserted (line 5-8, 11-14). The barrier can be implemented by using CUDA Cooperative Groups [1] or extending a lock-free GPU synchronization technique [48].

```

1 for i in range(10):
2   inpi = inp[i]
3   xi = matmul(inpi, wx)
4   h = tanh(xi + h)

```

(a) A simplified RNN model

```

1 __device__ void LoopUProg(float* inp, float* wx
  , float* h_in => float* h_out | float* tmp) {
2   float *inpi = tmp, *xi = tmp + 1024;
3   CopyUOp(h_in => h_out); Barrier();
4   for (int i = 0; i < 10; i++) {
5     GatherUOp(inp, &i => inpi); Barrier();
6     MatmulUOp(inpi, wx => xi); Barrier();
7     AddTanhUOp(xi, h_out => h_out); Barrier();
8   }
9 }

```

(b) Loop-*uTask* for the RNN model

Figure 10: Example of Loop-*uTask*

The Function-*uProgram* allocates the storage for Tensor *y* (line 4,10) and wraps the code with function name and signature (line 2). The `__device__` function qualifier is used so that this function can be called by other *uTasks*. We will omit the *uTask\_id* in the following sections and only show the generated code of one *uTask* inside the *uOperator* for brevity.

**Block alignment** One challenge of scheduling multiple DNN operators into a single GPU kernel comes from the variance of thread count inside each GPU block (*blockDim*). The *blockDim* of the kernel for a *uProgram* have to be set to the maximum *blockDim* of its *uOperators*, so that kernels with a large number of GPU blocks (*gridDim*) and small *blockDim* will execute inefficiently when they are scheduled into the same kernel of an operator with large *blockDim*. To address this problem, we re-implement the *uOperators* with configurable *blockDim* if possible (e.g., element-wise ones, reduction, and transpose). During schedule, COCKTAILER collects the fastest kernel of *uOperators* with predefined *blockDim* (e.g., *matmul* and convolution), and configure the *blockDim* of configurable *uOperators* to the maximum *blockDim* of the collected *uOperators*. If the *blockDim* of the collected *uOperators* varies greatly, COCKTAILER will leverage an extended Roller [57] to re-generate kernels with a fixed *blockDim*.

**Register pressure** The generated long-running GPU kernel may face register pressure. To alleviate this problem, COCKTAILER uses the profiling in §3.2 to detect performance drop due to register overuse and stop enlarging the current kernel. For control flow graph with no back edges, COCKTAILER can also utilize the branch recluster technique in §4.1.3 to both schedule the control flow to the accelerator side and reduce the kernel size.

### 4.1.2 Code Generation for Loop-*uTask*

**Overall structure** Figure 10(a) shows a simplified RNN model. It is scheduled to a Loop-*uProgram* with several Loop-*uTasks*. Each Loop-*uTask* in the *uProgram* contains *body\_uTasks* from three types of *uOperators*, i.e., gather,

```

1  if (cond) :
2    tmp1 = matmul(x, w1)
3    y = sigmoid(tmp1)
4    z = conv(y, w2)
5  else:
6    z = x + b

```

(a) A DNN model with branch

```

1  __device__ void BranchUProg(bool* cond, float*
   x, float* w1, float* w2, float* b, float*
   y_in => float* y_out, float* z_out | float*
   tmp) {
2  if (*cond) {
3    float *tmp1 = tmp;
4    MatmulUOp(x, w1 => tmp1); Barrier();
5    SigmoidUOp(tmp1 => y_out); Barrier();
6    ConvUOp(y_out, w2 => z_out);
7  } else {
8    AddUOp(x, b => z_out); // no Barrier
9    CopyUOp(y_in => y_out);
10 }
11 }

```

(b) Branch-*u*Task for the DNN model

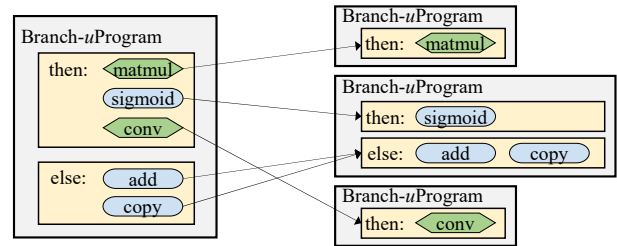
Figure 11: Example of Branch-*u*Task

matmul, and fused add-tanh operations. This Loop-*u*Program takes three input tensors named *inp*, *wx*, *h* (*h<sub>in</sub>* in Figure 10(b)), and produces an updated tensor *h* (*h<sub>out</sub>* in Figure 10(b)). The generated code of each Loop-*u*Task contains a loop (line 4) and the *u*Tasks (line 5-7) separated by barriers for synchronization across GPU blocks.

**Memory management** Different from existing DNN frameworks that allocate tensors at runtime, COCKTAILER needs to statically allocate tensor memory to execute the control flow operations on GPUs. The variables in the loop body can be divided into four categories: (1) constants (*wx*, *inp*); (2) intermediate results (*inp<sub>i</sub>*, *xi*); (3) iteration count (*i*); (4) loop-carried dependencies (*h*). All these variables are represented by pointers to the corresponding pre-allocated tensors and can be obtained from *get\_input\_data*. Specifically, the pointer to the constants are the corresponding function inputs, the intermediate results are allocated from a *tmp* buffer (line 2 in Figure 10(b)), and the pointer to the iteration count is *&i*. The pointers to the loop-carried dependencies are a little complex because the variable exists in both input tensors and output tensors of the Loop-*u*Operator. First, some CopyUOperators are inserted to copy the input tensors (*h<sub>in</sub>*) to the corresponding output tensors (*h<sub>out</sub>*). Then, the *body\_uTasks* is generated via only visiting the output tensors. Additional CopyUOperators and dependencies between *u*Operators in the loop body are added to ensure the correctness of the overlapped input and output tensors.

### 4.1.3 Code Generation for Branch-*u*Task

**Overall structure** Figure 11(a) contains a DNN model with two branches, The then branch takes tensors *x*, *w1*, and *w2* as inputs and produces tensors *y* and *z*; The else branch takes tensors *x* and *b* as inputs and produces tensor *z*. The input of the generated Branch-*u*Program is the union of inputs of



(a) Single kernel

(b) Branch reclustering

Figure 12: Optimize Branch-*u*Program by branch reclustering

the two branches as well as the *cond* tensor. The output is the union of the outputs of the two branches. If an output only exists in one branch, CopyUOperators will be added to the other branch to move the corresponding old value to the output tensor (line 9 of Figure 11(b)). The intermediate results are saved in tensors allocated from the *tmp* buffer. As only one branch may be executed in each run, the intermediate results of the two branches can use the same memory space.

**Branch reclustering** Scheduling a whole ControlFlow-*u*Program to a single GPU kernel is not always the best choice because different operations prefer different GPU occupancy (number of threads concurrently executed on an SM). For example, *matmul* uses a large amount of shared memory and registers for saving the tiles, resulting in limited occupancy, while element-wise operations prefer large occupancy to improve memory bandwidth. COCKTAILER also tries to schedule a Branch-*u*Program to multiple Branch-*u*Programs with each Branch-*u*Program containing *u*Operators with similar preferred occupancy and keeps the execution of branch condition on the GPU. The example model in Figure 12 contains limited-occupancy-*u*Operators *matmul* and *conv* (in green) and large-occupancy-*u*Operators *sigmoid*, *add*, and *copy* (in blue). These *u*Operators are scheduled into three Branch-*u*Programs for limited occupancy, large occupancy, and limited occupancy, respectively. The two branches are co-scheduled so each GPU kernel can contain *u*Operators from both branches. This branch reclustering technique reduces the kernel size, thus can also alleviate register pressure of large GPU kernels.

### 4.1.4 Code Generation for *u*Task Reference

**Overall structure** *u*Task reference is a special case that calls a *u*Task defined in another *u*Program. It is designed for recursions where a function may call its callers like Figure 13(a). To support recursion, the function declarations of all *u*Programs whose *u*Tasks are referenced by *u*Task references are generated at the start of the code (line 1 of Figure 13(b)). Then, all *u*Programs generate their function definitions. The maximum stack depth of our recursion implementation cannot be increased at runtime, so users need to manually set a limit to the stack depth, or COCKTAILER will use all free memory to save the intermediate results in the call stack. The base



```

1 def Recursion(l, r, is_leaf, inp, w, root):
2   cond = is_leaf[root]
3   if cond:
4     output = inp[root]
5   else:
6     a = Recursion(l, r, is_leaf, inp, w, l[root])
7     b = Recursion(l, r, is_leaf, inp, w, r[root])
8     c = a + b
9     output = matmul(c, w)
10  return output

```

(a) A recursive model

```

1 __device__ void RecursionUProg(float* l, float*
  r, bool* is_leaf, float* inp, float* w, int*
  root => float* output | char* tmp);
2 __device__ void BranchUProg(float* cond, float*
  l, float* r, bool* is_leaf, float* inp,
  float* w, int* root => float* output | char*
  tmp) {
3   if (*cond) {
4     GatherUOp(inp, root => output);
5   } else {
6     float *a = tmp, *b = tmp+256, *c = tmp+512;
7     RecursionUProg(l, r, is_leaf, inp, w, l + (*
  root) => a | tmp + 768); Barrier();
8     RecursionUProg(l, r, is_leaf, inp, w, r + (*
  root) => b | tmp + 768); Barrier();
9     AddUOp(a, b => c); Barrier();
10    MatmulUOp(c, w => output);
11  }
12 }
13 __device__ void RecursionUProg(float* l, float*
  r, bool* is_leaf, float* inp, float* w, int*
  root => float* output | char* tmp) {
14  float *cond = tmp;
15  GatherUOp(is_leaf, root => cond); Barrier();
16  BranchUProg(cond, l, r, is_leaf, inp, w, root
  => output | tmp + 256);
17 }

```

(b) The generated code

Figure 13: Example of recursion with  $\mu$ Task reference

case check is kept in the function body as a branch operation.

**Simulation of GPU stack** Though NVIDIA GPUs have the built-in support of recursion, the stack is slow and with very limited supported depth. The reason is that GPU needs to save the registers of all threads during function calls. However, in a DNN program, we only need to save the pointers to tensors and the program counter of the current stack frame before performing a function call. Moreover, the same set of tensor pointers are shared by multiple  $\mu$ Tasks, and only a single copy needs to be saved. Therefore, we have the opportunity to reduce the size of saved information to both increase the stack depth and reduce the time for saving the stack frame.

To achieve this, COCKTAILER implements a stack in global memory to simulate the function call behavior. As it is dangerous to directly update the program counter, COCKTAILER choose to inline all  $\mu$ Programs to a single function and use “goto” together with “labels” inserted into the inlined function to simulate the update of the program counter. The labels are placed at the start of the function and at the end of each function call inside the function. Instead of maintaining program counters, the stack saves the label of each stack frame. Each stack frame only consumes tens of bytes of memory, so COCKTAILER can also save the stack in GPU shared memory

Model	Input shape	Description
LSTM	64, BS, 256	hidden 256, length 64, layer 10
NASRNN	1000, BS, 256	hidden 256, length 1000, layer 1
Attention	BS, 12, 64, 64	head 12, hidden 768, length 64
Seq2seq	BS, 256	hidden 256, embed 3797 $\times$ 256, max length: 50 dataset: tatoeba-eng-fra
BlockDrop	BS, 3, 32, 32	drop layers from ResNet-32, dataset: CIFAR-10
SkipNet	BS, 3, 224, 224	drop layers from ResNet-101, dataset: ImageNet
RAE	127, 512	hidden 512, dataset: Stanford Sentiment Treebank

Table 1: Model configurations. BS refers to “batch size”.

to avoid the memory fence and inter-block barrier for maintaining a synchronized stack across different  $\mu$ Tasks when possible.

## 4.2 COCKTAILER on AMD ROCm GPUs

AMD ROCm GPUs provide a HIP programming model [2], which is similar to CUDA and is compatible with most CUDA statements. Besides, AMD provides a hipify tool to convert a CUDA kernel to a HIP kernel. COCKTAILER first generates a CUDA kernel and then leverages the hipify tool to convert it to the HIP version. Some  $\mu$ Operators are re-implemented due to the difference between CUDA and ROCm architectures.

## 5 Evaluation

**Platform** Our evaluation is on two accelerators: (1) NVIDIA Tesla V100-PCIE-32GB GPU with 2 Intel Xeon 5218 CPUs. The compiler is CUDA 11.5. (2) AMD Instinct MI100 GPU with 2 Intel Xeon 6338 CPUs. The compiler is ROCM 4.3.

**Baselines** We compare COCKTAILER with representative state-of-the-art deep learning frameworks including the most popular imperative framework PyTorch [36] v1.11 for CUDA and v1.10 for ROCM with TorchScript [3] enabled, the representative DAG-based framework TensorFlow v1.15 [4], and JAX v0.3.20 [11] with just-in-time compilation (JIT) enabled. ROCM 5.3 is used in JAX due to compatibility problems. Note that the latest TensorFlow 2 is redesigned as an imperative framework like PyTorch and JAX, therefore we choose TensorFlow v1.15 to evaluate the DAG-based framework. We also create a baseline that accelerates each basic block of the DNN program with Rammer [28] and relies on PyTorch for executing the control flow operations (COCKTAILERBASE). COCKTAILERBASE uses the same kernel implementation of each operator and the same compilation passes excluding the control-flow-related ones as COCKTAILER.

**Benchmarks** Our evaluation includes a set of representative DNN models that covers typical architectures like CNN, RNN, and transformers, different application domains including CV, NLP, and speech, and different types of control flow operations including loops, branches, and recursions. LSTM [16] is a representative RNN model for NLP and speech, and has been manually optimized by both deep learning frameworks and libraries. We use the built-in LSTM operators when possible, which are linked to the manually optimized LSTM im-

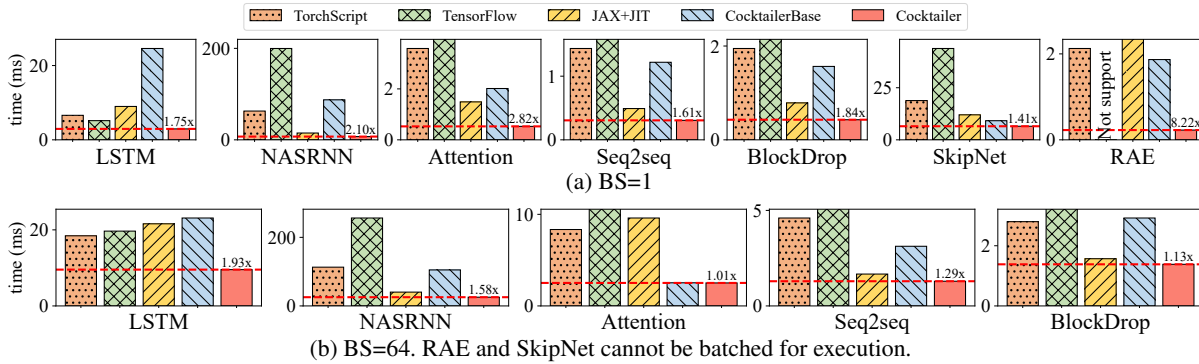


Figure 14: End-to-end DNN inference on NVIDIA V100 GPU

plementation in vendor libraries like cuDNN. NASRNN [58] is another RNN-based model created by network architecture search (NAS) that has not been manually optimized yet. Attention [43] is a widely used architecture in NLP and CV. We use an auto-regressive attention mechanism to continue sentences. The above three models contain loops with fixed iteration counts. Seq2seq [40] is a sequence-generation model that contains a while loop for continuously generating new tokens until an end-of-sequence (EOS) token is emitted or the maximum sequence length limit is reached. BlockDrop [47] and SkipNet [46] are two CNN-based CV models with branches for skipping some layers. Recursive Autoencoder (RAE) [39] is a well-known recursive model for NLP. The configuration of these models is listed in Table 1.

We set the batch size (BS) of the experiments to 1 and 64 to match the requirements for online inference and offline inference. The time is measured by averaging 100 tests after 100 warm-up runs. For models using real datasets, we randomly sample  $100 \times BS$  cases from the datasets.

## 5.1 End-to-end Evaluation on NVIDIA GPU

Figure 14 shows the inference performance of COCKTAILER by comparing with TorchScript, TensorFlow, and JAX with JIT enabled. All three frameworks support control flow operations by executing them on CPU. Overall, COCKTAILER outperforms the best baseline in each model by  $1.85\times$  in geometric mean (up to  $8.22\times$ ). Specifically, COCKTAILER outperforms TorchScript by  $3.98\times$  on average (up to  $9.35\times$ ), TensorFlow by  $18.45\times$  on average (up to  $196.85\times$ ), and JAX by  $3.05\times$  on average (up to  $327.62\times$ ). The time for compiling each model (except kernel tuning by AutoTVM and Anso) is several seconds to minutes.

**Models with loops** LSTM has been manually optimized by many frameworks and vendor libraries, and we use the fastest built-in implementation in the baselines. The core control flow operations of LSTM are two loops iterating over the input sequence and the layers respectively. TensorFlow and TorchScript use the manually-optimized LSTM in cuDNN library, while JAX loops over manually-optimized LSTM cell implementation. According to profiling, TensorFlow uses

the persistent-RNN [8] to optimize the loop over the input sequence, but it does not accelerate the loop iterating over the layers. TensorFlow with BS=64, TorchScript, and JAX only optimizes the operators in one LSTM cell, and does not perform joint optimizations on LSTM cells in different iterations. Different from these systems, COCKTAILER fully unrolls the static loop over layers and unrolls some steps of the loop over inputs, so that it can expose a large set of operators to the data flow optimization passes and benefit from the inter-operator schedule of Rammer. COCKTAILER outperforms all framework with handly-optimized implementations by  $1.75\times$  when BS=1 and  $1.93\times$  when BS=64.

The computation of NASRNN and Attention has not been manually optimized. These frameworks optimize the basic block using only passes for compiling static data flow, and execute the loop on CPU. COCKTAILER performs some loop optimizations and schedules the loop to thread block level. With such optimizations, COCKTAILER achieves  $2.10\times$  on NASRNN model and  $2.82\times$  speedup on Attention model over the fastest baseline when BS=1. However, COCKTAILER only achieves  $1.01\times$  speedup over COCKTAILERBASE on Attention BS=64 because control flow only take a small portion of execution time when the body computation is large enough.

Seq2seq is implemented with a while loop, and existing frameworks need to copy the decision from the accelerator to the CPU to decide whether to continue the loop. By executing the loop on GPU, COCKTAILER can both use fewer kernels and avoid such synchronization. The speedup over the fastest baseline is  $1.61\times$  and  $1.29\times$  when BS=1 and BS=64, respectively.

**Models with branches** BlockDrop and SkipNet drop some layers from ResNet with decisions generated at runtime. The baselines need to copy the decision from GPU to CPU to decide whether to launch the next layer. COCKTAILER avoids such synchronized copy by scheduling the branch to block level for BlockDrop BS=1, and using branch reclustering for BlockDrop BS=64 and SkipNet BS=1. COCKTAILER accelerates BlockDrop by  $1.84\times$  and  $1.13\times$  over the best baseline when BS=1 and BS=64, respectively, and accelerates SkipNet by  $1.41\times$ .

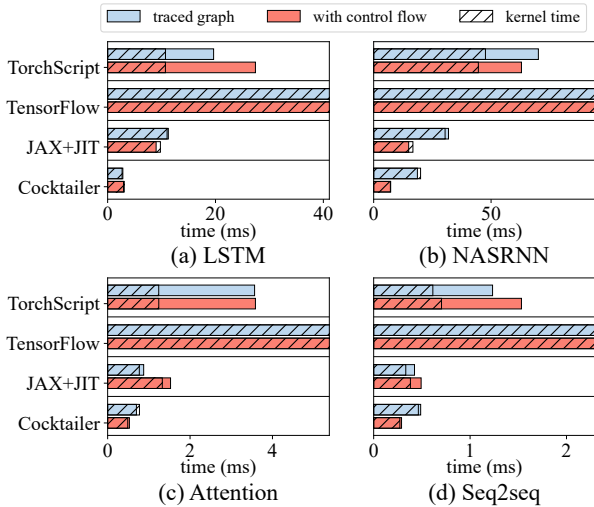


Figure 15: Control flow overhead of models with loops.

**Model with recursion** RAE is a recursive model. TensorFlow does not support recursion. PyTorch and JAX can only run this model in Python, resulting in poor performance. COCKTAILER schedules the recursion to block level with parallel execution and executes the recursive calls efficiently with the simulated stack, resulting in  $9.35\times$ ,  $327.62\times$ , and  $8.22\times$  speedup over PyTorch, JAX, and COCKTAILERBASE respectively.

**Discussion** Whether a model is control flow bound or data flow bound depends on the ratio of control flow computation and data flow computation. According to the evaluation among different models in Figure 14, it is clear that COCKTAILER can achieve higher speedup when model execution has more control flow computation, e.g., NASRNN, RAE. When the data flow occupies the most computation (e.g., Attention in BS=64), COCKTAILER can achieve similar performance with the fastest baseline.

## 5.2 Control Flow Overhead Analysis

In this section, we evaluate the performance degradation caused by control flow boundary in different systems when BS=1. The results are shown in Figure 15, 16, and 18. For each model, we choose an input with a typical execution trace of the dataset. We compare the real scenario that executes control flow at runtime to executing the traced computation graph with no control flow to evaluate the overhead. The traced graph baseline of COCKTAILER is compiled by Rammer with the same kernel implementations and compilation passes for data flow as COCKTAILER.

**Models with loops** Figure 15 shows the control flow overhead of models with loops. The input data of LSTM, NASRNN, and Attention is a sequence with length provided in Table 1, and the input to Seq2seq generates a 10-token-sequence which is near to the average sequence length of the dataset.

For LSTM model, Rammer can explore the parallelism of

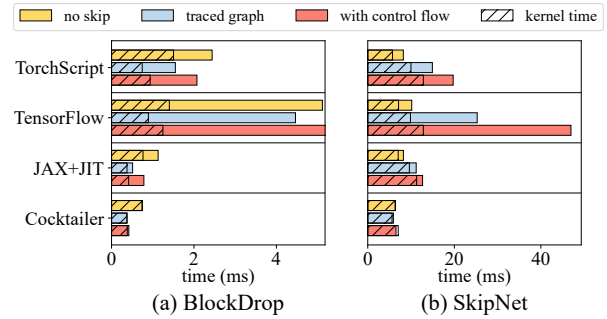


Figure 16: Control flow overhead of models with branches. "No skip" refers to running all layers of the ResNet model.

cells in different steps if all steps are unrolled. The dynamic unrolling of COCKTAILER provides similar performance with Rammer, but can support dynamic step count. Other systems do not explore such parallelism and are slower than COCKTAILER. To expose the loop of TorchScript and TensorFlow, we do not use their cuDNN LSTM here.

For NASRNN, Attention, and Seq2seq models, COCKTAILER schedules the loop to thread block level with only one GPU kernel, and is faster than Rammer which uses a larger number of kernels. A similar phenomenon also appeared in the NASRNN model with JAX. JAX generates thousands of different kernels for execution the unrolled loop and is slower than looping over the NASRNN cell with 3 kernels for 1000 times. This indicates that an efficient implementation of control flow can sometimes be faster than running the unrolled data flow.

For Seq2seq model, TorchScript, TensorFlow, and JAX need to copy the decision back to CPU to decide whether to execute the next iteration of the while loop, causing a synchronization between CPU and GPU. Therefore, when control flow is used, the increase of execution time is larger than that of kernel time. COCKTAILER does not have such a problem because all control flow operations are executed on GPU.

**Models with branches** Figure 16 shows the control flow overhead of models with branches. The two models skip some layers from a ResNet model, and we add a "no skip" which is a normal ResNet without skipping layers. The ratios of executed layers are 7/15 for BlockDrop and 23/33 for SkipNet, which are similar to the average ratio of the models respectively.

Due to the synchronization between CPU and GPU, the control flow operations of the baselines increase the execution time by at least 34% over the traced version for BlockDrop, while COCKTAILER only increases the execution time by 11%. Therefore, though more than half of the layers are skipped, the performance improvement of layer skipping compared with the original ResNet model is only at most  $1.44\times$  in the baselines, while COCKTAILER achieves  $1.79\times$  speedup. In SkipNet, the network for making the skip decision is heavier and the ratio of executed layers is larger, so the traced graph may take longer execution time than the original ResNet model. The slow execution of control flow makes the performance of

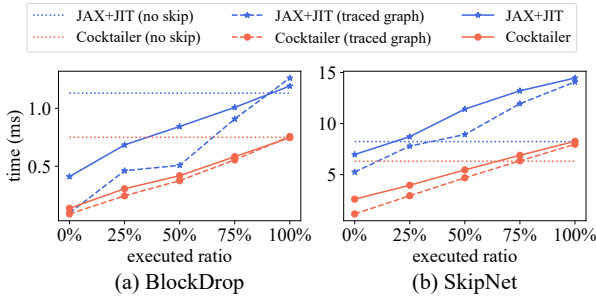


Figure 17: Different ratio of executed layers

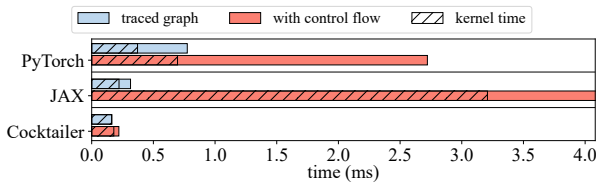


Figure 18: Control flow overhead of RAE with recursion.

this model even worse in baselines, while COCKTAILER can still provide reasonable performance.

Figure 17 shows the performance of BlockDrop and SkipNet at different ratios of executed layers. The results of JAX, the fastest baseline of the two models are also included. When the executed ratio is 0, the model executes all control flow operations but runs no layers, and COCKTAILER achieves  $3.00\times$  and  $2.68\times$  speedup over JAX on BlockDrop and SkipNet, respectively. This proves the low control flow overhead of COCKTAILER. In SkipNet, if the model is executed with JAX, the layer-skipping can improve the performance only when the ratio of executed layers is lower than 20%, while if executed with COCKTAILER, this ratio becomes about 65%.

**Model with recursion** Figure 18 shows the control flow overhead of the recursive RAE model. The input is a 65-node tree from the Stanford Sentiment Treebank dataset. PyTorch and JAX can only execute the recursion in Python and the time is much longer than executing the traced graph. Rammer processes nodes without dependencies in parallel with a static schedule that only works for this tree, while COCKTAILER executes the model by control flow operations on the GPU side and only increases the time by 11%.

**Discussion** Compared with the traced graph baseline which removes all the control flow operations in the models and can be considered as the optimal status, COCKTAILER achieves similar performance. Besides, the overall latency of COCKTAILER is similar to the kernel time, which indicates that COCKTAILER can minimize the overheads introduced by control flow. Furthermore, the evaluations on BlockDrop and SkipNet show that COCKTAILER also enables scenarios like efficient computation by achieving real speedup. We hope COCKTAILER can provide more flexibility for algorithm researchers to design DNN architectures with control flow.

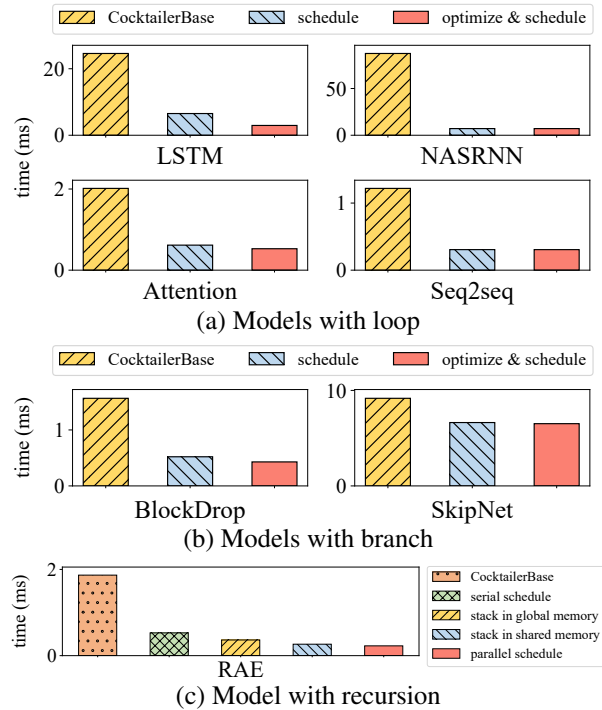


Figure 19: Breakdown of models with BS=1

### 5.3 Breakdown of Optimizations

Figure 19(a) provides the breakdown of optimizations applied on models with loops. On average, scheduling the loop to block level provides  $4.95\times$  speedup over COCKTAILERBASE that executes the loop in PyTorch runtime. And applying the optimizations in §3.2, especially the dynamic loop unrolling further improves the performance of LSTM by  $2.22\times$  and Attention by  $1.17\times$ . In LSTM, the loop is re-scheduled to kernel level after loop unrolling.

Figure 19(b) provides the breakdown for models with branches. The branches of the two models are executed on GPU, with branch reclustering used in SkipNet. The scheduling provides  $3.01\times$  and  $1.38\times$  speedup over COCKTAILERBASE on BlockDrop and SkipNet, and the optimizations further accelerate the two models by  $1.21\times$  and  $1.02\times$ .

Figure 19(c) shows the performance of the RAE model. Executing the recursion on GPU provides  $3.54\times$  speedup over COCKTAILERBASE. The simulation of stack using global memory and shared memory are  $1.45\times$  and  $1.99\times$  faster than using the built-in GPU stack. And the parallel scheduling of  $\mu$ Programs further improves the performance by  $1.17\times$ .

### 5.4 End-to-end Evaluation on AMD GPU

Figure 20 compares TorchScript, TensorFlow, JAX with JIT enabled and COCKTAILER on AMD MI100 GPU with BS=1. COCKTAILER outperforms the three frameworks on all benchmarks by  $2.97\times$  over TorchScript on average (up to  $5.86\times$ ),  $21.28\times$  over TensorFlow on average (up to  $112.34\times$ ), and  $3.22\times$  over JAX on average (up to  $272.63\times$ ).



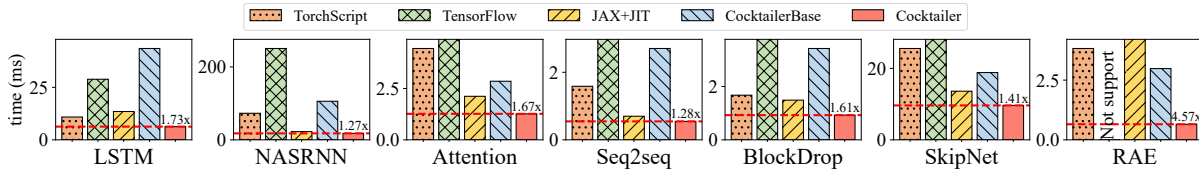


Figure 20: End-to-end DNN inference on AMD MI100 GPU with BS=1

## 6 Related Work

Supporting control flow in deep learning can be divided into two categories. The first one, represented by TensorFlow 1.x [4] and TorchScript [36], executes control flow operations in the framework runtime on CPU. Control flow is implemented as special operators (`NextIteration` for loops [52], `Switch` for branches [52], and `InvokeOp` for recursions [19]) or instructions in the runtime. The second one, represented by Chainer [42], PyTorch [36], and JAX [11], leverages the runtime of general-purpose language like Python to support the control flow operations. The control flow operations are expressed with Python statements and executed by the Python interpreter. AutoGraph [31], Janus [18], and Terra [22] show that the control flow operations expressed by general-purpose languages can sometimes be converted to the control flow operators in the framework runtime. Despite different ways of supporting control flow, the control flow operations in these works can only be executed by CPU.

Some special forms of control flow have been deeply optimized. VersaPipe [55] optimized pipelines for general GPU programs. Cortex [10] provides interfaces to describe recursion with data patterns (i.e., the recursion tree structure). It assumes that the jump direction of all control flow only depends on the input recursion tree structure, so it does not apply to control flow depending on dynamically computed data, e.g., the while loop with unknown iteration count in Seq2seq [40], and the branches whose direction is decided at runtime in BlockDrop [47] and SkipNet [46]. COCKTAILER does not assume the availability of such tree structures and works on these models.

Past works on batching (e.g., DyNet [33], Cavs [49], Tensorflow Fold [27], BatchMaker [12], Program-counter-autobatching [37], and ORCA [51]) enable the parallelization in different control flow operations by introducing a scheduler to batch the ready-to-execute operators, which is another applicable approach and is complementary to COCKTAILER. Specifically, COCKTAILER can compile subgraphs of a model, and then batching can be applied to these subgraphs. Applying batching on the more coarse-grained subgraph granularity can also reduce the scheduling cost in the batching scheduling.

There are many deep learning compilers for optimizing a computation graph without control flow, including TVM [6], TASO [20], Rammer [28], DNNFusion [35], PET [44], and AStitch [56]. These optimizations are compatible with COCKTAILER. COCKTAILER even enlarges their optimization scope because the boundary of control flow has been reduced. Com-

pilation optimizations like function inline [5], loop unroll [9] have been introduced in general-purpose language compilers on CPU programs and have been implemented in COCKTAILER. COCKTAILER further introduces the new *uTask* abstraction to represent both data flow and control flow operations, which aligns with the parallelism of hardware accelerators, enabling analyzing and optimizing both data flow and control flow computation over heterogeneous accelerators (i.e., GPU). To scale DNN models on distributed architectures, frameworks and compilers like Tofu [45], FlexFlow [21], GSPMD [50], PipeDream [32], Tutel [17], FasterMOE [14], FlexMoE [34], BaGuaLu [29], Alpa [54] and SuperScaler [25] parallelize the execution of deep learning models across multiple hardware devices, but only focus on models with static architectures or specific types of dynamic models (e.g., Mixture-of-Experts [30]). COCKTAILER exposes the parallelism of control flow operations, which can be leveraged to support dynamic models over distributed devices.

## 7 Conclusion

DNN frameworks and compilers suffer from performance issues when supporting sophisticated dynamic DNN models. The parallelism mismatch between control flow and data flow results in separate execution of DNNs on the CPU and accelerator, causing not only overheads but also missed optimization opportunities. COCKTAILER supports sophisticated DNN models by co-scheduling the execution of control flow and data flow that (1) provides the fine-grained *uTask* abstraction for control flow and data flow in DNN programs to open a holistic scheduling space on hardware accelerators; (2) designs the scheduling mechanism and a heuristic policy to exploit this scheduling space; (3) provides control flow optimizations in both scheduling and code generation. Evaluations demonstrate that COCKTAILER significantly outperforms state-of-the-arts on sophisticated DNN models. By enabling the co-optimizing of control flow and data flow in a single space, COCKTAILER positions itself as a new enhancement to the deep learning infrastructure.

## Acknowledgements

We thank the anonymous reviewers and our shepherd, Prof. Wenjun Hu, for their extensive suggestions. This work is partially supported by National Key R&D Program of China under Grant 2021ZD0110104, National Natural Science Foundation of China (62225206).

## References

- [1] Cooperative Groups. <https://devblogs.nvidia.com/cooperative-groups/>.
- [2] HIP Programming Guide. [https://rocmdocs.amd.com/en/latest/Programming\\_Guides/HIP-GUIDE.html](https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html).
- [3] TorchScript. <https://pytorch.org/docs/stable/jit.html>.
- [4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.
- [5] Pohua P Chang and W-W Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, 1989.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.
- [7] An-Chieh Cheng, Chieh Hubert Lin, Da-Cheng Juan, Wei Wei, and Min Sun. Instanas: Instance-aware neural architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3577–3584, 2020.
- [8] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Adam Coates Mike Chrzanowski, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 16)*, pages 2024–2033, 2016.
- [9] Jack J Dongarra and A\_R Hinds. Unrolling loops in fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.
- [10] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.
- [11] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.
- [12] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.
- [13] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021.
- [14] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 120–134, 2022.
- [15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
- [17] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *arXiv preprint arXiv:2206.03382*, 2022.
- [18] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.
- [19] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.
- [20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the*

27th ACM Symposium on Operating Systems Principles, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

- [21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.
- [22] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeong-In Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. *Advances in Neural Information Processing Systems*, 34, 2021.
- [23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [24] Yanwei Li, Lin Song, Yukang Chen, Zeming Li, Xiangyu Zhang, Xingang Wang, and Jian Sun. Learning dynamic routing for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8553–8562, 2020.
- [25] Zhiqi Lin, Youshan Miao, Guodong Liu, Xiaoxiang Shi, Quanlu Zhang, Fan Yang, Saeed Maleki, Yi Zhu, Xu Cao, Cheng Li, et al. SuperScaler: Supporting flexible dnn parallelization via a unified abstraction. *arXiv preprint arXiv:2301.08984*, 2023.
- [26] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [27] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.
- [28] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.
- [29] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, et al. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 192–204, 2022.
- [30] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *The Artificial Intelligence Review*, 42(2):275, 2014.
- [31] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Autograph: Imperative-style coding with graph-based performance. volume 1, pages 389–405, 2019.
- [32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [33] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. *Advances in Neural Information Processing Systems*, 30, 2017.
- [34] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flex-MoE: Scaling large-scale sparse pre-trained model training via dynamic device placement. *arXiv preprint arXiv:2304.03946*, 2023.
- [35] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.
- [36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.
- [37] Alexey Radul, Brian Patton, Dougal Maclaurin, Matthew Hoffman, and Rif A Saurous. Automatically batching control-intensive programs for modern accelerators. *Proceedings of Machine Learning and Systems*, 2:390–399, 2020.
- [38] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.
- [39] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, pages 151–161, 2011.

- [40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems, NIPS'14*, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.
- [41] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.
- [42] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.
- [43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [44] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.
- [45] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.
- [46] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.
- [47] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8817–8826, 2018.
- [48] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.
- [49] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, 2018.
- [50] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.
- [51] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [52] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys '18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [53] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.
- [54] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.
- [55] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599. IEEE, 2017.
- [56] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Pro-*



*gramming Languages and Operating Systems*, pages 359–373, 2022.

- [57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.
- [58] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

## A Artifact Appendix

### Abstract

This artifact helps to reproduce the results of OSDI'23 paper: COCKTAILER: Analyzing and Optimizing Dynamic Control Flow in Deep Learning.

### Usage

The input of COCKTAILER is a PyTorch program. COCKTAILER exports the PyTorch program to ONNX format with ONNX loop and branch operators as well as an extended invoke operator for recursion. Then COCKTAILER generates the code with optimizations described in the paper and wraps the code to a PyTorch custom operator for execution.

### Scope

The artifact can be used to reproduce the experiments of the paper, including the end-to-end comparison (Figure 14 and 20), control flow overhead analysis (Figure 2, 15, 16, and 18), performance of different ratio of executed layers (Figure 17), and breakdown of optimizations (Figure 19).

### Contents

This artifact includes the code of COCKTAILER, input data of experiments, a guide for setting up the environment of the experiments, and scripts for running the experiments. It helps to reproduce the following Figures:

- Figure 2: Control flow overhead in JAX
- Figure 14: End-to-end DNN inference on NVIDIA V100 GPU
- Figure 15: Control flow overhead of models with loops
- Figure 16: Control flow overhead of models with branches
- Figure 17: Different ratio of executed layers
- Figure 18: Control flow overhead of RAE with recursion
- Figure 19: Breakdown of models with BS=1
- Figure 20: End-to-end DNN inference on AMD MI100 GPU with BS=1

### Hosting

The main contents of COCKTAILER are hosted at [https://github.com/microsoft/nfusion/tree/cocktailer\\_artifact/artifacts](https://github.com/microsoft/nfusion/tree/cocktailer_artifact/artifacts), branch `cocktailer_artifact`.

## Requirements

This artifact needs two machines:

- a machine with 8 NVIDIA V100 GPUs, with NVIDIA driver properly installed. Users can either follow the installation guide to setup the software environment or install the NVIDIA Container Toolkit to reproduce the results within the docker provided by the artifact.
- a machine with 1 AMD MI100 GPU, with ROCm driver and docker properly installed. Users can then reproduce the results within the dockers provided by the artifact.





# WELDER: Scheduling Deep Learning Memory Access via *Tile-graph*

Yining Shi<sup>†\*</sup> Zhi Yang<sup>†</sup> Jilong Xue<sup>◇</sup> Lingxiao Ma<sup>◇</sup> Yuqing Xia<sup>◇</sup>  
Ziming Miao<sup>◇</sup> Yuxiao Guo<sup>◇</sup> Fan Yang<sup>◇</sup> Lidong Zhou<sup>◇</sup>  
<sup>†</sup>*Peking University*    <sup>◇</sup>*Microsoft Research*

## Abstract

With the growing demand for processing higher fidelity data and the use of faster computing cores in newer hardware accelerators, modern deep neural networks (DNNs) are becoming increasingly memory intensive. A disparity between underutilized computing cores and saturated memory bandwidth has been observed in various popular DNN models. This inefficiency is caused by both the conventional treatment of DNNs as compute-intensive workloads and the lack of holistic memory access optimization in DNN models.

In this paper, we introduce WELDER, a deep learning compiler that optimizes the execution efficiency from a holistic memory access perspective. The core of WELDER is *tile-graph*, an abstraction that facilitates fine-grained data management at tile level. By leveraging the observation of optimization independence across memory layers, WELDER is able to decompose the whole combinatorial DNN optimization space into several independent ones and effectively trade off between intra- and inter-operator data reuse using a tile traffic-based cost model. This allows WELDER to unify previous ad-hoc memory optimizations into a single space, generate efficient execution plans with 89 more optimization patterns, and outperform state-of-the-art solutions significantly. WELDER is also able to handle DNN models with arbitrarily large input by combining the existing accelerator memory and host memory as a whole system.

## 1 Introduction

Deep neural networks (DNNs) have been used in a wide range of tasks like vision and language analysis and synthesis. Conventional wisdom treats DNNs as compute-intensive workloads. A DNN model is often defined as a dataflow graph (DFG), where each node represents a compute-intensive operator (e.g., matrix multiplication). These operators are offloaded to modern accelerators with massive parallel computing cores, such as GPUs and TPUs [23], to speed up com-

putation. To utilize accelerators efficiently, DNN frameworks and compilers explore various optimization techniques, such as code specialization [15, 50, 52] and operator fusion [15, 31].

Although these computation centric optimizations are shown effective for classic DNN models, we observe that modern DNNs are becoming increasingly memory intensive. Our profiling on a range of state-of-the-art DNN models reveals that the bottleneck of the end-to-end DNN computation is mostly on GPU memory. The memory bandwidth utilization can be as high as 96.7% while the average utilization of computing cores is only 51.6% (§2). Moreover, we observe the disparity between the underutilized cores and the saturated memory bandwidth could become even larger with the evolution of both hardware and DNN models. Modern models are processing higher fidelity data, e.g., larger images, longer sentences, high-definition graphics, which consume more memory bandwidth in the computation. Furthermore, the faster computing cores (e.g., TensorCore [6]) impose an even greater pressure on memory.

Optimizing memory intensive DNN workloads is challenging as it requires improving the sophisticated data access and reuse patterns across multiple memory layers (e.g., GPU DRAM and shared memory). From the memory perspective, DNN computation comprises of a repetitive process for each operator to 1) load input tensors across memory hierarchy, 2) compute at the cores, and 3) store the resulting tensors across memory hierarchy. To derive a good data access pattern, it requires a careful calculation of the size of tile, a partition of a tensor, along each tile dimension. Such a tiling strategy is already difficult to obtain in existing practice [5, 50, 52]. As a further complication, due to the different algorithmic semantics, each operator may require a different data access patterns. Such diversity across operators makes *inter-operator* data reuse especially challenging, and often infeasible. If the derived tile shape of an operator at a certain memory layer does not match that of a downstream operator, it is difficult to reuse the tile at that layer. Consequently, existing approaches either focus on intra-operator optimization and leave all inter-operator intermediate tensors in the lowest memory layer

\*Work is done during the internship at Microsoft Research.



(e.g., GPU memory), or rely on rule-based operator fusions to alleviate the inter-operator memory overhead. These rules are only applicable for specific operator combinations (e.g., register fusion for element-wise operators [10, 13, 15], shared memory fusion for a limited set of operator types [51]) and can be suboptimal when having different input sizes or running on different hardware configurations.

In this paper, we introduce WELDER, a deep learning compiler that holistically optimizes memory access for end-to-end DNN models consisting of general operators. The design of WELDER is based on three key observations. First, to resolve potential tile shape conflicts between two adjacent operators, we observe that their aligned tile shape can be automatically inferred by propagating an output tile shape from back to front, given that the computing logic in each operator can be accurately preserved (e.g., through the tensor expression). Second, to decide which tile shape will lead to better performance, by enforcing the computation pattern to be aligned with hardware feature (e.g., TensorCore), we can just minimize the data traffic across all memory layers. Given the operators with aligned tile configuration, we notice that their data traffic can be easily modeled based on their input/output tile sizes and the input/output tensor shapes. Finally, when considering the whole memory hierarchy, we observe that the optimization of memory traffic is inherently independent across memory layers, i.e., *inter-layer independence*. Particularly, the above traffic model is determined only by the tile configuration at the memory layer of interest. These observations allow us to optimize the whole space with an effective process: starting from aligning two adjacent operators at independent memory layers, deciding their optimal tiling size at the right memory layer guided by traffic costs, and expanding the optimization to include further operators.

WELDER incorporates these insights into a new DNN compiler design. First, to facilitate fine-grained data management, WELDER proposes *tile-graph*, a tile-level data-flow graph to model DNN computation. Each node in the graph processes one data tile of a tensor at a time. To map DNN computation to a multi-layered memory hierarchy, WELDER allows the control of each node’s data tile size and the desired memory layer to reuse the data tile between two nodes. Specifically, WELDER provides a `SetConnect` interface to set the data reuse layer for each edge and a `Propagate` interface to infer the tile configurations within a group of connected nodes. Second, to efficiently optimize the tile level data-flow scheduling holistically, WELDER exploits the *inter-layer independence* properties in the data-flow computation to decouple the optimization space into multiple sub-spaces. Based on this, WELDER proposes a two-layered scheduling policy that enumerates different memory connection options for each edge and decides on an efficient tile configuration for each sub-space guided by the traffic cost model. Finally, the optimized execution plan is mapped to executable code for a specific hardware accelerator through four abstracted com-

puting interfaces defined in the hardware layer: `Allocate`, `LoadTiles`, `ComputeTile`, and `StoreTiles`.

With the tile level holistic data-flow scheduling, WELDER is the first to unify all common operator fusions (e.g., register-based element-wise fusion, shared-memory fusion, etc.) into a single framework. This generality allows WELDER to find 89 uncommon operator fusion patterns automatically that are mostly unexplored by existing rule-based approaches (§5.2). Interestingly, our approach can easily support new requirements for handling DNN models with arbitrarily large input (e.g., high-resolution images), where even a single operator may be too large to fit in the GPU memory. Specifically, by extending the current memory hierarchy with additional layers (e.g., host memory), WELDER can generate an optimized execution plan across the combined hierarchy of host and device memory.

We have implemented WELDER on top of TVM [15], Rammer [31] and Roller [52]. Our evaluation is conducted on 10 state-of-the-art DNN models covering both classic and recent model structures for various tasks including vision, NLP, 3D-graphics, etc. The evaluation results show that WELDER significantly outperforms the state-of-the-art DNN framework and compilers like PyTorch, ONNXRuntime, and Anso on both NVIDIA and AMD GPUs, with up to  $21.4\times$ ,  $8.7\times$ ,  $2.8\times$  speedups, respectively. WELDER’s automatic optimization even outperforms TensorRT [7] and Faster Transformer [2], which are a highly optimized handcrafted DNN inference library and a model-specific implementation from NVIDIA, with up to  $3.0\times$  and  $1.7\times$  speedups. Furthermore, when running these models on hardware with faster computing cores such as TensorCore, we observe a larger improvement in performance, highlighting the importance of memory optimization for future AI accelerators.

## 2 Motivation

**Modern DNNs are memory-bounded.** Figure 1 presents the average GPU utilization, including both computational FLOPS and global memory throughput, for a representative DNN benchmark running with ONNXRuntime [8]. As shown, the average computation utilization is only 51.6% while memory utilization is 96.7%. When examining the model types, we find that ResNet and BERT, which are dominated by convolution and matrix multiplication operators and can achieve relatively high computation utilization (e.g., >80%), are two representative classical models. However, the remaining models, which are popular models proposed in recent years, exhibit low computation efficiency due to introducing more memory-intensive patterns beyond compute-intensive operators. Additionally, we observe that the new DNN models often have a higher ratio of memory store traffic to load traffic compared to classical models. The primary reason is these models tend to process high-fidelity data and generate large activations across layers. However, current systems such

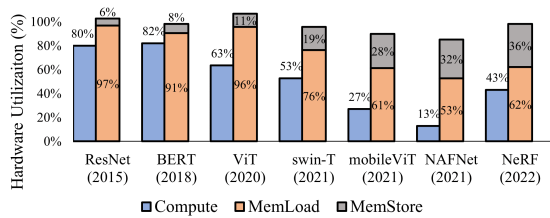


Figure 1: Computation FLOPS and memory bandwidth utilization for different models on NVIDIA V100 GPU.

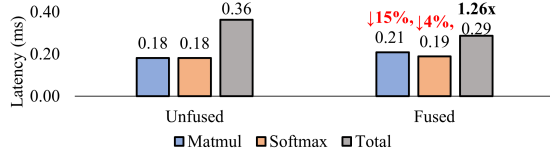


Figure 2: Latency numbers of unfused, fused, and each individual kernels of Matmul and Softmax.

as ONNXRuntime have limited optimizations for reducing inter-operator traffic. This indicates that these models will frequently exchange large intermediate data across operators through global memory. The results highlight the need for optimizing memory access efficiency across operators.

### Conflicted intra- and inter-operator data reuse patterns.

Optimizing intra-operator and inter-operator data reuse simultaneously is challenging. An operator is often implemented as nested multi-level loops over all tensor dimensions. Within the operator, the data reuse across multiple memory layers are often implicitly optimized using sophisticated loop tiling techniques [5, 50, 52]. We consider a typical pattern of two consecutive operators, i.e., Matmul and Softmax. When the two operators are optimized independently, their optimal tile sizes in shared memory are different, e.g.,  $[32 \times 64]$  for Matmul and  $[4 \times 128]$  for Softmax. As a result, Softmax is unable to reuse the intermediate data from Matmul in shared memory, leading to a total latency of 0.36ms, as shown in Figure 2. However, if we force them to take into account both intra- and inter-operator data reuse, the fused operator latency can be reduced to 0.29ms, achieving a 1.26x speedup. Upon examining their aligned tile size (i.e.,  $[16 \times 128]$ ), we observe that both operators sacrifice their own efficiency (e.g., with 15% and 4% performance degradation when running separately, due to suboptimal data tile for intra-operator data reuse) in favor of overall efficiency. This demonstrates the need for an efficient data reuse solution across intra-operator and inter-operator to optimize memory access holistically.

**Key observations.** Through a further analysis on the example in Figure 2, we have identified three key observations. First, an aligned tile configuration across operators can be deduced based on a chain of shape inference starting from an output tile shape. For example, if we want to compute a  $[4 \times 128]$  output tile of Softmax, based on its computing logic (e.g., tensor expression), we can deduce that its dependent

input tile shape is also  $[4 \times 128]$ . Then, by using  $[4 \times 128]$  as the output tile of Matmul, we can further deduce that input tile shapes of Matmul will be  $[4 \times k]$  and  $[k \times 128]$ , where  $k$  is an reduction size that can be set as any number not exceeding the reduction dimension size of the Matmul. In this way, the two operators can be fused by reusing the intermediate data tile ( $[4 \times 128]$ ) in shared memory.

Second, given the aligned tile configuration and the original tensor shapes, the total memory traffic can be easily derived analytically. In this example, the Matmul takes input tensors A in shape  $[98304 \times 64]$  and B in  $[64 \times 128]$  respectively, and an output tensor C in  $[98304 \times 128]$ . The Softmax then takes C as input and produces an output tensor D in the same shape. Input tensors A, B, and the output tensor D are in global memory. Given these shapes, we can first calculate the memory traffic when computing a single output tile (i.e.,  $[4 \times 128]$ ) of tensor D. To do so, it will first load a tile of shape  $[4 \times k]$  from tensor A and a  $[k \times 128]$  tile from tensor B for Matmul, and then the intermediate tile  $[4 \times 128]$  will be consumed by Softmax in shared memory, and write a tile of shape  $[4 \times 128]$  to tensor D, where the  $k$  can be replaced as 64 given the input tensor shape of  $[98304 \times 64]$ . Thus, the total traffic incurred in global memory for an individual output tile is 35KB  $((4 \times 64 + 64 \times 128 + 4 \times 128) \times 4 \text{Bytes}(\text{FP32}))$ , where the traffic of the intermediate tile  $[4 \times 128]$  is saved due to data reuse in shared memory. To compute the full output tensor D, a total of 24,576 such computations are required (i.e.,  $(98304 \times 128) / (4 \times 128)$ ), resulting in a total global memory traffic of 840MB (i.e.,  $24,576 \times 35\text{KB}$ ). Interestingly, changing the output tile to  $[16 \times 128]$  will reduce the total traffic to only 264MB, following the same calculation.

Finally, our traffic-cost calculation is only determined by the tile configuration at the memory layer of interest, e.g., the output tile shapes of  $[4 \times 128]$  or  $[16 \times 128]$  in shared memory, once the tensor shapes are specified. This allows us to choose the tile size for each layer independently in order to optimize the traffic cost from the lower memory layers.

These observations together provide us an effective way to optimize memory access holistically, i.e., aligning a group of adjacent operators through an output tile shape, deciding on the best tile shape based on memory traffic, and optimizing for each memory layer independently. In this way, WELDER is able to change the original coarse-grained inter-operator dependency into a more fine-grained tile-level dependency, which essentially removes some false barriers between operators and enables more concurrency.

## 3 WELDER Design

The observations in §2 motivate WELDER, a deep learning compiler that aims to improve the performance of modern DNNs in a holistic memory access scheduling space. Figure 3 shows the system overview. WELDER takes a full DNN model as input and converts it into a data-flow graph of tile-based

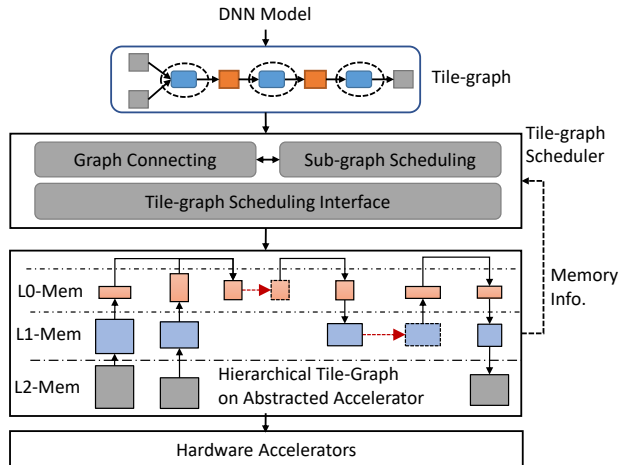


Figure 3: System overview of WELDER.

computing tasks (i.e., *operator-tiles*), which is called *tile-graph* (§3.1). A tile-graph provides fine-grained control over data tile configurations and memory placement. Given a tile-graph, WELDER resolves the intra-operator and inter-operator data-reuse conflicts through a "first-connect-then-schedule" approach: it first assumes two adjacent operators can reuse data tile at a certain memory layer (i.e., connect), and then derives the best common tile shape to see if the total memory traffic can be reduced. To facilitate this goal, WELDER provides two tile-graph scheduling interfaces: *SetConnect* and *Propagate* (for the chain of shape inference). Based on this, we propose a two-step scheduling algorithm, i.e., *graph connecting* and *sub-graph scheduling*, to recursively decide an efficient tile-graph execution plan for multiple memory layers, known as a *hierarchical tile-graph* (§3.2). Finally, this plan is then mapped to an executable code for a specific hardware accelerator using four abstracted computing interfaces defined in the hardware layer, i.e., *Allocate*, *LoadTiles*, *ComputeTile*, and *StoreTiles* (§3.3). The memory specification of the abstracted accelerator is used by the tile-graph scheduling layer to guide the optimization process.

### 3.1 Operator-tile and Tile-graph

WELDER defines DNN computation in a fined-grained task granularity named *operator-tile*. A DNN operator, such as convolution, can be implemented as multiple homogeneous operator-tiles, which are executed either in a streaming or parallel manner to compute all the data tiles in the output tensors [31]. Each operator-tile takes as input a data tile sliced from the input tensors and computes a data tile in the output tensors, with the computing logic described by an index-based tensor expression [15]. Figure 4(a) and (b) shows examples of operator-tiles for Conv and MaxPool, where the Conv operator computes a  $[1 \times 1 \times C]$  data tile by taking a  $[3 \times 3 \times C]$  data tile as input, and the MaxPool operator takes an input tile of  $[2 \times 2 \times F]$  and computes an output tile of  $[1 \times 1 \times F]$ .

To improve the utilization of hierarchical memory re-

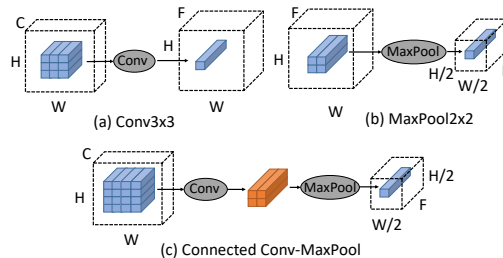


Figure 4: Illustration of two operator-tiles: (a) Conv and (b) MaxPool; and (c) connecting them into a tile-graph (the weight tensor of Conv is omitted for simplicity).

sources, such as the shared memory, WELDER allows two adjacent operator-tiles to be "connected" through a common intermediate data tile, also known as a *reuse-tile*. This allows the second operator-tile to consume the data produced by the first operator-tile directly, without the need to materialize it into a full intermediate tensor. Figure 4(c) illustrates an example of this connection between two operator-tiles for Conv and MaxPool, using a  $[2 \times 2 \times F]$  reuse-tile. Multiple operator-tiles can be connected along each adjacent edge to form a data flow graph of operator-tiles, known as a *tile-graph*.

**Tile propagation.** Once connected, most tiles in a tile-graph are correlated, which can be automatically inferred by propagating an output tile shape to the entire graph. This is achieved by using a chain of shape inferences from the output nodes to the inputs. For each operator-tile, the dependent region of the input tensor can be accurately determined by analyzing its tensor expression and output tile size. In cases where the input region may contain irregular patterns such as sparse or noncontinuous access (e.g., *Gather* or *Convolution* with strides), our expression analysis provides a conservative upper bound as the input tile shape. If the tile-graph has multiple output nodes, their output shapes may also be correlated, as they may share a common ancestor node in the graph. In this case, after propagating the first output tile, we propagate separate shapes for the remaining output nodes, aligning them with the first one. If there is an inconsistent tile shape between the two propagations, we do not connect the latter output node to the current graph.

**Memory traffic and footprint.** After the tile propagation, the memory traffic and footprint of a tile-graph can be determined. First, the memory traffic for an individual tile-graph can be calculated by summing its input and output tile sizes. The total traffic is obtained through further multiplying this value by the number of tile-graphs needed to compute the full output tensor (e.g., through dividing the tensor size by the output tile size). Second, the minimum memory footprint for the tile-graph can be calculated using a memory allocation algorithm (e.g., *bestfit* [19]) by allocating all data tiles in a topological order. As a footprint optimization, input tiles that contain reduction axes can be further partitioned into smaller

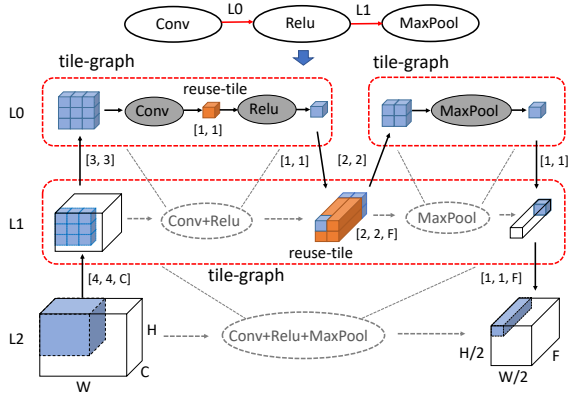


Figure 5: Map three consecutive operators to a three-layer memory hierarchy (the weight of Conv is omitted).

ones, which can be loaded and consumed sequentially by accumulating their results to the output tile. Specifically, a particular policy can automatically try different tiling sizes along the reduction axes during the tile propagation.

### 3.2 Tile-graph Scheduling

To map a DNN model represented by an initial data flow graph to an accelerator, we can recursively partition each operator into multiple operator-tiles to fit within each memory layer, and connect operator-tiles at higher memory layers to exploit inter-operator data reuse. As a result, an entire DNN computation can be modeled as a data streaming pipeline over a two-dimensional space, with data tiles moving up and down the memory hierarchy vertically and being passed to successor operators at different layers horizontally.

Figure 5 illustrates an example of mapping three consecutive operators (Conv, ReLU, and MaxPool) to a three-layered memory hierarchy (e.g., from L2 to L0). The input tile of the Conv operator is repeatedly loaded from L2 to L1 and then L0 for computation. By connecting the Conv and ReLU operators at L0, the output of the Conv operator can be reused as the input for the ReLU operator, and the two operators form a tile-graph at L0. At the same time, they are consolidated into a virtual node (i.e., Conv+ReLU) in L1. The output of the ReLU is then continuously spilled into the data tile at L1 and reused as the input for the MaxPool, through further connection at L1. This allows all three operators to form a single tile-graph at the L1 layer, resulting in the virtual node Conv+ReLU+MaxPool in L2. After this recursive process, all operators are connected at the lowest layer as a single tile-graph.

**Decoupling optimization space.** Given the observation that DNN computation is mostly memory-bounded, our major optimization goal of the data streaming pipeline can be transformed to minimizing the memory traffic. This allows us to decompose the whole optimization space into several sub-spaces by leveraging the inherent independence of optimizing

```
void SetConnect(Edge *edge, MemLevel level);
TileConfig Propagate(TileGraph g,
                    Map<Axis, Dim> config);
size_t MemFootprint(TileGraph g);
size_t MemTraffic(TileGraph g);
```

Figure 6: The scheduling interface in WELDER

traffic across memory layers. Specifically, the total data traffic loaded from and stored to a lower memory layer for a given tile-graph can be estimated by just its output tile shape, i.e., used to deduce all the input and output tile shapes. Based on this property, different tile-graphs from the same or different memory layers can independently optimize their memory traffic by searching for the optimal tile shapes. For example, in Figure 5, the tile-graph of Conv and ReLU at L0 can be optimized independently of the L1 tile-graph (e.g., formed by the Conv+ReLU and MaxPool operators), which is referred to as *inter-layer independence*. This further implies that the optimal tile configurations of the sub-graphs Conv-ReLU and MaxPool at L0 are also independent, due to their independence with the tile-graphs at L1, to which we refer as *intra-layer independence*. In practice, the only constraint is that the tile size at the lower memory level must be larger than the tile size at the upper memory level. This is often the case, as the lower memory level typically has greater capacity than the upper memory level. With these properties, we can independently schedule each tile-graph given a graph connection plan.

**Scheduling interface.** WELDER provides two scheduling interfaces to control graph connecting and sub-graph tiling, as shown in Figure 6. First, the graph connecting is implemented using the SetConnect interface, which assigns a memory level for an edge in the tile-graph (the lowest level by default). After connecting, the tile shapes in the graph is inferred through the Propagate interface, by specifying the dimensional sizes of the output tiles and the optional reduction axes in input tiles. For example, in Figure 5, we can use the SetConnect interface to connect Conv and ReLU at L0 and connect ReLU and MaxPool at L1. After the connection, for the sub-graph Conv+ReLU, we can use the Propagate to infer the intermediate reuse-tile shape (i.e., [1, 1]) by specifying the output tile shape of [1, 1]. Similarly, we can also infer the intermediate reuse-tile shape of sub-graph Conv+ReLU+MaxPool (i.e., [2, 2, F]) by specifying the output tile shape of [1, 1, F]. The two scheduling primitives are essentially two interfaces to update the edges and vertices of the tile-graph. Particularly, SetConnect is used to add a connection between two nodes and Propagate is used to set tile configuration for a node. They together form a complete interface for updating the tile-graph. Note that these primitives are only used by WELDER’s scheduling policy and transparent to the end users. WELDER also provides two cost interfaces, MemFootprint and MemTraffic, to calculate the memory



```

1 Func GraphConnecting(g:Graph, d:Device):
2   for node : TopologySort(g.nodes()) do
3     for edge : node.out_edges() do
4       for level : d.MemLevels() do
5         SetConnect(edge, level);
6         s = ExtractSubgraph(node, 0);
7         configs = SubGraphTiling(s, 0, tensor_shapes);
8         if t = Min(d.Profile(configs)) < best_latency
9           best_latency = t;
10          best_level = level;
11        SetConnect(edge, best_level);
12 Func SubGraphTiling(g:Graph, level:Memory, c: Config)
13   configs = PriorityQueue();
14   for subtile : EnumerateSubtiles(g, c) do
15     config = Propagate(g, subtile);
16     if MemFootprint(g) > level.capacity
17       continue;
18     configs.push(config, priority=MemTraffic(g));
19   results = Dict();
20   for config : TopK(configs, k) do
21     // return empty sub-graph at top level to exit recursion
22     subgraphs = unique([ExtractSubgraph(node, level+1)
23                       for node in g.nodes()]);
24     for subgraph : subgraphs do
25       subgraph_configs = SubGraphTiling(subgraph,
26   level+1, config);
27       results[config].append(subgraph_configs);
28   Return results;
29 Func ExtractSubgraph(node:Node, level:Memory)
30   nodes = Set();
31   for edge : node.InOutEdges() do
32     if edge.connect_level > level
33       nodes.insert(ExtractSubgraph(edge.node, level));
34   return SubGraph(nodes);

```

Figure 7: Two-step tile-graph scheduling algorithm.

footprint and the total traffic of a tile-graph, which serve as our cost models to guide the scheduling.

**Scheduling policy.** WELDER adopts a two-step scheduling algorithm to optimize data flow computation effectively. Specifically, a *graph-connecting scheduler* first enumerates different graph connecting plans by setting different memory reuse levels for each edge, and then a *sub-graph scheduler* quickly searches for efficient tile configurations for each sub-graph decoupled by the graph-connecting scheduler. Figure 7 shows the two-step scheduling algorithm in WELDER. First, given a DNN data flow graph  $g$  and an accelerator device  $d$ , the graph-connecting scheduler enumerates all graph nodes and their output edges in a topological order (line 1-3). For each edge, WELDER tries different connection levels (e.g., using the `SetConnect` interface) (line 5). It then extracts the connected sub-graphs where all edges have connection

Allocate	Allocate workspace in a memory layer
LoadTiles	Load input tiles from lower memory layer
ComputeTile	Compute an operator-tile at the top layer
StoreTiles	Store result tiles back to lower memory layer
MemLevels	Query memory hierarchy configurations

Table 1: Device interfaces in abstracted hardware accelerator.

levels higher than 0. Here, we use the number 0 to represent the lowest memory level, and larger numbers for higher levels. The `ExtractSubgraph` function is implemented in line 26-31. For the extracted sub-graph, WELDER calls the `SubGraphTiling` function to get several efficient tile configurations and chooses the optimal one by profiling on the hardware (line 7-10). After comparing with all other connection levels, WELDER sets the best connection level for the current edge.

Then, the sub-graph scheduler (i.e., the `SubGraphTiling` function) takes as input a sub-graph and the last level tile configuration and searches for efficient tile configurations for the current level. First, WELDER enumerates the tile sizes (i.e., `EnumerateSubtiles` in line 14) for output dimensions using a tile shape expanding approach similar to Roller [52], which enlarges initial tile shape (e.g., size of 1) towards the shapes that can reduce total traffic and align with hardware features. After getting the output tile shapes, we can infer the complete tile configuration using the `Propagate` interface and check if it exceeds the memory capacity using the `MemFootprint` interface, or appends it to a sorted result list with the memory traffic as the key (e.g., using the `MemTraffic` interface) (line 15-18). Finally, we choose the top  $K$  configurations with the least memory traffic for the current level, and then extract the upper-level sub-graphs and decide their best tile configurations recursively by calling `ExtractSubgraph` and `SubGraphTiling` (line 20-24).

Note that WELDER has no assumption on the memory size on different memory hierarchies, as our scheduling policy can always try its best to determine the optimal layer and tile size to place intermediate data, so as to minimize the overall latency. While WELDER always favors hardware with large higher-level fast memory (e.g., shared memory) that can hold a sufficiently large intermediate data tile, because too small tile sizes could lead to worse intra-operator data reuse. The scheduling result of a data flow graph in WELDER is a *hierarchical tile-graph*, which starts as a full graph at the lowest memory level and is recursively split into several sub-graphs in the upper layers, all the way to the top level.

### 3.3 Mapping to Hardware Accelerator

The hierarchical tile-graph generated by WELDER is an abstracted execution plan that can be mapped to an executable code for a specific hardware accelerator. To facilitate this mapping, WELDER provides an abstracted accelerator device with hierarchical memory layers. The memory configura-

```

void ExecuteGraph(TileGraph g, MemLevel level,
                 void *in, void *out) {
    void *mem = Allocate(g.MemFootprint(), level);
    LoadTiles(in, mem);
    for (auto n : g.nodes())
        if (level == MemLevel.top)
            ComputeTile(n, mem.in[n], mem.out[n]);
        else
            ExecuteGraph(n.TileGraph(), level+1,
                        mem.in[n], mem.out[n]);
    StoreTiles(mem, out);
}
// execute a full DNN graph at memory level 0
ExecuteGraph(graph, 0, inputs, outputs);

```

Figure 8: Compilation routine of hierarchical tile-graph.

tions, such as the number of layers, memory capacity, and transaction width of each layer, can be obtained through a `MemLevels` interface (e.g., used in Figure 7). With this abstracted memory layer, it is easy to extend an existing accelerator with additional memory layers (e.g., host memory or SSD) as a new device, allowing it to handle very large tensors that may not fit in the single device memory (§5.4 for more details). WELDER’s performance gain mainly comes from the bandwidth gap between memory layers. Thus, as long as a lower-level memory becomes the bottleneck and a high-level memory can hold the intermediate data tile, WELDER can automatically pipeline the inter-operator data transfer on the faster, high-level memory.

In order to execute a hierarchical tile-graph on a hardware accelerator, WELDER provides four computing interfaces: `Allocate`, `LoadTiles`, `ComputeTile`, and `StoreTiles` (listed in Table 1). The routine for executing a hierarchical tile-graph using these interfaces is shown in Figure 8. The process starts by executing the bottom-layer tile-graph (i.e., the full DNN graph). For each tile-graph, it first allocates the necessary workspace in the corresponding memory layer (using the `Allocate` interface) and loads the input tiles into this space (`LoadTiles`). Then, it executes all the nodes in the sub-graph in a topological order. If the current memory layer is the top level, the node is executed directly in the computing cores (`ComputeTile`). Otherwise, the execution of the upper-level tile-graph is called recursively. Finally, the result tiles in the current space are stored in the lower memory layer (`StoreTiles`). This execution routine can be used as both a code generation process or a runtime process, depending on whether a specific accelerator implements these computing interfaces as code emitters or executable function calls. In WELDER, they are currently implemented as code emitters to generate the accelerator-specific computing logic. By executing this recursive routine, the entire hierarchical tile-graph is unrolled and a full-model computation program with all the necessary optimizations is generated automatically.

## 4 Implementation

WELDER is implemented based on open-source DNN compilers, TVM [15], Roller [52] and Rammer [31]. It leverages TVM for writing kernel schedule, Roller for enumerating efficient tile configurations, and Rammer for the end-to-end graph optimization. WELDER’s core mechanisms, including the tile-graph, tile propagation, scheduling algorithm, code generation, etc., are implemented in 5.2k lines of code. WELDER takes an ONNX graph as input and performs common graph optimizations such as constant folding and simple element-wise fusion. It then converts the optimized graph into a tile-graph for holistic memory scheduling optimization. WELDER is implemented on both CUDA and ROCm GPUs, and GraphCore IPU through the unified device interface (Table 1). For CUDA and ROCm GPUs, WELDER schedules data tiles on three memory layers: global memory (DRAM), shared memory, and register. To handle large images on CUDA GPUs and GraphCore IPU, we also extend their device memory by adding a host memory layer.

### 4.1 Hardware-aligned Tile Search

**Enumerate efficient data tile size.** WELDER takes into account several hardware-related factors that could impact the data access efficiency by introducing a penalty factor to the traffic cost model. First, if there is uncoalesced memory access, the total memory traffic will include the additional transactions required for these accesses. For instance, in CUDA GPUs, it is always preferable to use coalesced memory access for a contiguous 128 bytes of data (one transaction). Second, when there is insufficient parallelism due to a large tile size, the memory traffic is increased proportionally based on the utilization percentage of the computing cores. Finally, we add an infinite penalty if the total memory footprint of a given tile configuration exceeds the memory capacity. To avoid enumerating inefficient candidates, WELDER searches for output tiles by only enumerating the dimensions that can reduce traffic the most according to the cost model, and retrieves only top  $k$  candidates with the minimum traffic.

**Decide aligned computation parallelism.** In GPUs, the top-level operator-tiles that are executed in the same thread-blocks must agree on a unified block size (e.g., number of threads). To ensure this alignment, WELDER first enforces sufficient parallel tiles at the register level to align with the hardware parallelism (i.e., by enumerating hardware-aligned tiles). For example, in NVIDIA V100 GPUs, the tile number should be greater than 128, as each SM has 4 warp schedulers and each warp has 32 threads. We then determine the greatest common divisor among the tile numbers of all operators as the common thread-block size, if it is larger than the hardware parallelism (e.g., 128) and less than the maximum limitation (e.g., 1024). Otherwise, we set the block size to a number

that equals the hardware parallelism. Once the block size is decided, we bind all operator-tiles at the register level to these threads. If a single thread needs to run multiple tiles, we use TVM’s virtual thread to bind them, thus allowing concurrent data access over all memory banks and avoiding bank conflicts.

**Support TensorCore.** WELDER leverages TensorCore to accelerate certain operators such as GEMM, BatchMatmul, and Convolution (using implicit GEMM [28]) on CUDA GPUs. We add annotations to these operators indicating which axes will be bound to CUDA’s Warp-Level Matrix Operations. For top-level operator tiles, we bind them to warps (instead of threads) to perform MMA operations. Additionally, we introduce some extra constraints when enumerating tile sizes, such as ensuring that the number of threads is an integral multiple of the warp size and that the axes (M, N, and K) in each tile are an integral multiple of the fragment size of the MMA operations.

## 4.2 Code Generation and Compilation

WELDER’s kernel generation is based on TVM. In particular, the register level tile connection is implemented using TVM’s `compute_inline` schedule primitive. For shared memory level connection, we only use TVM to generate standalone kernels for each connected part above the shared memory, and then apply several additional passes to compose these standalone kernels into a single fused kernel.

**Load/store rewriting.** The standalone kernels generated by TVM load and store data from global memory. We rewrite these global memory accesses to shared memory accesses by adding an additional TIR [11] pass to TVM’s lowering procedure. Additionally, we add memory fences to prevent race conditions and apply padding to handle bank conflicts in the buffers. As a result, the original global kernel can be transformed into a device function, which is included in the final fused kernel.

**Block/thread index remapping.** Some operators cannot be directly connected to others and require remapping of their `blockIdx` and `threadIdx` values. The `BlockIdx` remapping is used for operators such as `Transpose`. The remapping relationship is deduced from their tensor expressions. The `ThreadIdx` remapping is used to connect 2D thread blocks to 1D thread blocks. This is necessary when inter-thread reduction or TensorCore primitives require the use of a 2D thread block (both `threadIdx.x` and `threadIdx.y`), while others may use a 1D thread block (only `threadIdx.x`). A 2D thread block can be mapped to a 1D thread block as long as their total number of threads is equal.

**Memory management.** We manage all shared memory, including that allocated in each standalone kernel and the inter-

operator reuse buffer, in a uniform manner. First, we analyze the liveness of each buffer based on the topology execution order and convert them into a sequence of allocation and free operations. We then use the bestfit algorithm to compute the offset for each shared memory allocation, taking into account any alignment requirements for data types and TensorCore operations (e.g., aligning to 32 bytes to avoid misaligned address access).

**Compilation speedup.** WELDER optimizes the compilation speed through parallel compilations and sub-graph caching. First, by taking advantage of the independence between configurations, WELDER can use multi-processes to build and evaluate each configuration in parallel. Second, in most DNN models, some sub-graph patterns often repeat for multiple times. To avoid the redundant optimization, WELDER leverages a sub-graph signature to cache each unique graph pattern. For example, in a 12-layer BERT model, we can cache the optimization result (kernel code and profiled latency) for the first layer and reuse it for all the remaining 11 layers.

## 5 Evaluation

### 5.1 Experimental Setup

We evaluate WELDER using three servers equipped with different accelerators: NVIDIA GPU, AMD GPU, and Graphcore IPU. Two servers are equipped with the NVIDIA GPUs. The first one is an Azure NC24s\_v3 VM with Intel Xeon E5-2690v4 CPUs and NVIDIA Tesla V100 (16GB) GPUs, running on Ubuntu 16.04 with CUDA 11.0. The second one is a local workstation with Intel(R) Xeon(R) E5-2678 v3 CPUs and NVIDIA GeForce RTX 3090 GPUs, running on Ubuntu 18.04 with CUDA 11.3. The AMD GPU server is equipped with Intel Xeon CPU E5-2640 v4 CPU and AMD Radeon Instinct MI50 (16GB) GPUs, running on Ubuntu 18.04 with ROCm 5.2.3. The IPU server is an Azure ND40s\_v3 VM with Intel Xeon Platinum 8168 CPUs and 16 IPU with Poplar-sdk 3.0.

**DNN workloads.** WELDER is evaluated on 10 DNN models with different model types, including CNNs, Transformer, CNN-Transformer and multilayer perceptrons (MLP), and most of which are the state-of-art in the corresponding tasks. Table 2 characterizes them with a comparison of their model types, tasks, and the years of publication. For all models in the table, we use their official PyTorch implementations without modification.

**Baselines.** We compare WELDER with several DNN frameworks, including PyTorch (v1.12) [10] and ONNXRuntime (v1.12) [8], as well as state-of-the-art DNN compilers such as Ansor (v0.9) [50] and Rammer [31]. We also compare WELDER with TensorRT (v8.4) [7], a vendor-specific inference library for NVIDIA GPUs. For transformer models,

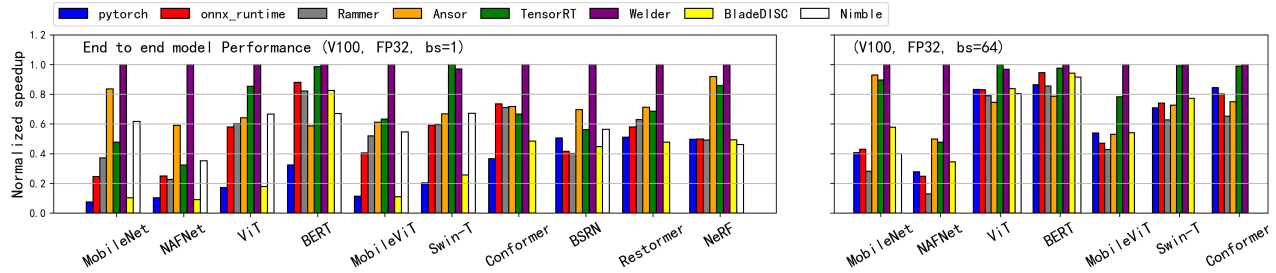


Figure 9: End-to-end model inference performance on NVIDIA V100 GPU (SIMT Core only). (left : batch size of 1, right : batch size of 64).

Model	Type	Task	Year
MobileNet [41]	CNN	Image Classification	2018
BERT [16]	Transformer	NLP	2018
ViT [17]	Transformer	Image Classification	2020
Conformer [20]	CNN+Transformer	Speech Recognition	2020
MobileViT [32]	CNN+Transformer	Image Classification	2021
Swin-Transformer [30]	Transformer	Image Classification	2021
NeRF [33]	MLP	3D-scene Generation	2021
NAFNet [14]	CNN	Image Restoration	2022
Restormer [48]	CNN+Transformer	Image Restoration	2022
BSRN [29]	CNN	Image Super-resolution	2022

Table 2: DNN models evaluated in WELDER.

we further compare WELDER with NVIDIA’s FasterTransformer (v5.2) [2], a hand-crafted C++ library optimized for transformer models. We also include BladeDISC (v0.3.0) [1] that implements the latest AStitch [51] for the kernel fusion optimization. We also include Nimble [25] which implements multi-stream scheduling as a baseline on NVIDIA GPUs.

To evaluate a model on these baselines, we first trace the model in PyTorch and export it to the ONNX format. We then use this ONNX model as input to other frameworks, including WELDER, Anso, ONNXRuntime, and TensorRT. For the ONNXRuntime, we use its CUDA execution provider and set its graph optimization level to "ALL" to achieve the best performance. For TensorRT, we use its Python API to build an engine for the input ONNX model. For Anso, we set the total number of tuning trials to 800× the number of tasks in each model. For all frameworks, we place the input and output tensors in GPU device memory to avoid additional data movement costs. During evaluation, we first perform some warm-up iterations and then run each workload repeatedly for at least 5 seconds. We only report the average speed for each model, as we observe very little variation in all cases. The average performance (e.g., speedup) across models is calculated by geometric mean in all experiments.

## 5.2 Evaluation on NVIDIA GPUs

This section answers the following questions: 1) How does WELDER perform in comparison with state-of-the-art DNN frameworks or compilers? 2) To what extent can WELDER further boost performance with TensorCores? 3) Can WELDER automatically discover new optimization patterns beyond previous expert-designed fusion rules? 4) How well does

WELDER improve both the memory and computational efficiency? 5) What is the search efficiency of WELDER’s holistic optimization?

**End-to-end performance.** Figure 9 shows the performance of WELDER and other baselines for batch size of 1, expressed as the normalized speedup relative to the best result. The geometric mean speedup that WELDER achieves over DNN frameworks is  $4.29\times$  for PyTorch and  $2.07\times$  for ONNXRuntime. PyTorch does not perform well for models with batch size 1 due to high Python overhead in its computation graph. In contrast, ONNXRuntime is a more optimized framework that removes Python overheads and implements pattern-based graph optimizations. WELDER also outperforms Rammer by  $1.96\times$ , as Rammer can only fuse independent parallel kernels instead of dependent ones through shared memory. When evaluating BladeDISC (implementing AStitch), we notice that it encounters "unsupported operator" failures and falls back to PyTorch runtime for the majority of models. For models without encountering any failure (including BERT, MobileNet, BSRN and NeRF), WELDER is  $2.70\times$  faster than BladeDISC. Regarding the Nimble baseline, WELDER achieves an average speedup of  $1.79\times$ , excluding the models where Nimble fails to execute.

Anso improves DNN performance by generating high-performance tensor programs and using rule-based fusion across operators at the register level (e.g., Matmul+BiasAdd, Conv2D+ReLU). However, it cannot exploit further memory reuse opportunities, leading to an average performance gap of  $1.44\times$  compared to WELDER. This is evident in CNN models such as NAFNet ( $1.70\times$ ) and BSRN ( $1.43\times$ ), which mainly consist of convolutions with relatively small channels that can be well optimized by WELDER. WELDER also outperforms Anso by a significant margin on Transformer-based models such as BERT ( $1.71\times$ ), Swin-Transformer ( $1.45\times$ ), and ViT ( $1.56\times$ ), due to Anso’s inability to fuse patterns like LayerNorm or Softmax in the attention block. Furthermore, WELDER performs well for CNN+Transformer models, achieving speedups of  $1.64\times$ ,  $1.39\times$ , and  $1.29\times$  on MobileViT, Conformer, and Restormer, respectively, as WELDER can cover fusion opportunities in both the CNN and Transformer parts of these models. We also observe that



WELDER only slightly outperforms Anso on NeRF (1.09 $\times$ ), mainly due to that the compute-intensive MLP dominates the computation without further optimization opportunities.

Finally, TensorRT is a specialized DNN inference library provided by NVIDIA with highly optimized operators. WELDER is comparable to TensorRT on popular transformer models such as BERT (1.02 $\times$ ) and Swin-T (0.97 $\times$ ). This is because TensorRT has incorporated expert-designed fusion rules and in-house kernels for some popular models, including transformer-based models, thereby leaving limited room for further optimization. In contrast, WELDER identifies optimization patterns automatically and achieves performance that is on par with TensorRT, despite relying on less performant kernels for compute-intensive operators. It is worth noting that kernel optimization is complementary to WELDER, and further optimized kernels may offer even greater benefits for WELDER. Additionally, for newer and more diverse models such as NAFNet, WELDER has demonstrated superior performance to TensorRT, with speedups of up to 3.09 $\times$  due to its generality. Overall, our system outperforms TensorRT with an average speedup of 1.47 $\times$ .

Figure 9 also shows the normalized performance for a larger batch size of 64. The last three models in Table 2 are unable to be traced on PyTorch with large batch sizes due to their use of large input size. Under this setting, WELDER continues to outperform all other baselines, providing an average speedup of 1.83 $\times$  over PyTorch, 1.90 $\times$  over ONNXRuntime, 2.1 $\times$  over Rammer, 1.57 $\times$  over BladeDISC, 1.49 $\times$  over Nimble, 1.47 $\times$  over Anso, and 1.21 $\times$  over TensorRT, respectively. We observe that for large batch sizes, frameworks using CUDA libraries perform much better, compared to the results for a batch size of 1. This leads to smaller speedups over PyTorch, ONNXRuntime, and TensorRT for WELDER, while the speedup over Anso remains similar to the results for a batch size of 1.

**Performance with TensorCore.** The faster computing throughput of TensorCore can put greater pressures on memory access. To understand the optimization behaviors when running on TensorCore, we convert our benchmark models (both weights and activations) to half-precision float type (FP16) with PyTorch, as TensorCore only supports FP16. This is done using the tools in the `onnxconverter_common` package [9], with the exception for TensorRT, which converts through its own converter as it often produces better results.

Figure 10 shows the performance comparisons of WELDER with other frameworks using TensorCore for batch sizes of 1 and 64. For the 10 cases that use a batch size of 1, WELDER outperforms PyTorch, ONNXRuntime, BladeDISC, Nimble, Rammer, and TensorRT. The averaged speedup is 7.18 $\times$  (up to 21.4 $\times$  on MobileNet) to PyTorch, 3.08 $\times$  (up to 8.72 $\times$  to on Conformer) to ONNXRuntime, 5.29 $\times$  (up to 16.9 $\times$  on MobileNet) to BladeDISC, 2.72 $\times$  (up to 5.58 $\times$  on NeRF) to Nimble, 2.76 $\times$  (up to 5.42 $\times$  on NAFNet) to Rammer, and

Model	DT	BS	WELDER(ms)	FT-CPP(ms)
BERT	FP32	1	3.13	3.15
BERT	FP32	64	118.6	119.8
BERT	FP16	1	1.49	1.50
BERT	FP16	64	24.82	22.29
ViT	FP32	1	1.33	1.96
ViT	FP32	64	15.29	15.68
ViT	FP16	1	1.09	1.89
ViT	FP16	64	4.79	5.15
swin-T	FP32	1	2.59	2.38
swin-T	FP32	64	66.13	72.62
swin-T	FP16	1	1.43	1.60
swin-T	FP16	64	23.12	28.67
<b>geometric mean</b>			6.71	7.46

Table 3: Performance for WELDER and FasterTransformer

1.53 $\times$  (up to 2.98 $\times$  on NAFNet) to TensorRT, respectively.

For the remaining 7 cases in Figure 10 that uses a batch size of 64, WELDER outperforms PyTorch by 1.98 $\times$ , ONNXRuntime by 2.13 $\times$ , BladeDISC by 1.97 $\times$ , Nimble by 3.84 $\times$ , Rammer by 3.45 $\times$  and TensorRT by 1.16 $\times$  respectively.

Some of the speedups are much larger than the ones achieved on SIMT cores. Especially for the NeRF model, WELDER outperforms TensorRT by 2.34 $\times$  on TensorCore, while the speedup on SIMT cores is only 1.16 $\times$ . This is mainly because TensorCore can greatly accelerate the compute-intensive part of the model, making the optimization of the remaining memory-intensive part more critical.

Note that Anso is not included in this experiment as it does not support TensorCore. For a fair comparison, we disable WELDER’s TensorCore feature and evaluate these FP16 models on SIMT cores by comparing with Anso in Figure 11. It shows a slightly higher speedups (1.74 $\times$  on average and up to 2.82 $\times$ ) compared with the ones in FP32.

**Performance on another NVIDIA GPU** We also conduct evaluations on RTX-3090, another widely-used GPU, which utilizes a distinct Ampere architecture. The RTX-3090 exhibits various new features compared to the V100, including advancements in memory load and TensorCore instructions, as well as a different number of streaming multiprocessors (SM). For the sake of conciseness, we solely compared WELDER with TensorRT on the RTX-3090, as TensorRT consistently delivers superior performance compared to other baselines on NVIDIA GPUs. The results, depicted in Figure 12, illustrate that WELDER outperforms TensorRT with an average speedup of 1.40 $\times$ , calculated using the geometric mean of all 34 test cases. Notably, this speedup is similar to the one observed on the V100 GPU, which amounted to 1.36 $\times$ , thereby highlighting WELDER’s adaptability across diverse GPU architectures.

**Patterns automatically discovered.** WELDER automatically discovers around 300 different fused subgraphs, which is counted by unique operator types under all 34 compiled test cases of the 10 models. Among them, 89 patterns contain at least two reduction-based operators which cannot be covered

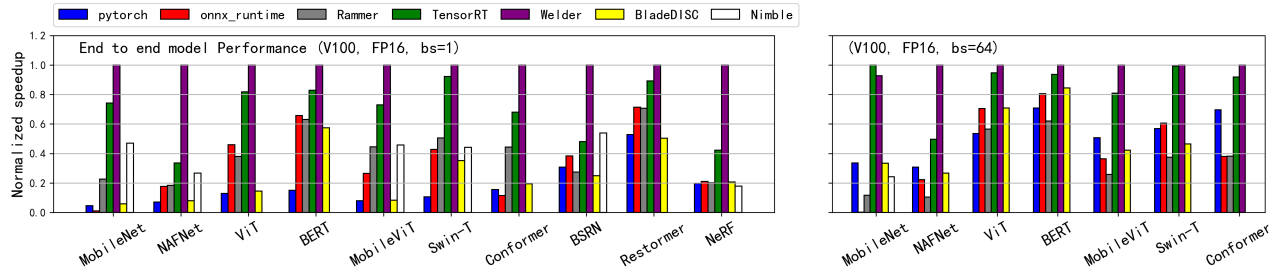


Figure 10: End-to-end model inference performance on NVIDIA V100 GPU (TensorCore enabled). (left : batch size of 1, right : batch size of 64).

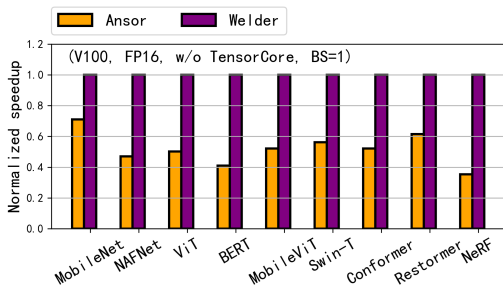


Figure 11: Comparing with AnsoR under FP16 w/o TensorCore

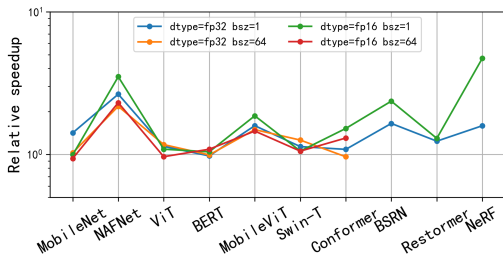


Figure 12: Comparing with TensorRT on NVIDIA RTX-3090 by simple element-wise fusion rule in AnsoR. To the best of our knowledge, many of these patterns are uncommon fusion patterns that have not been explored by manually-designed rules or automatic fusion optimizations. Figure 4 illustrates two examples of such patterns, which fuse multiple Convolution or MatMul (i.e., Dot) operators with other memory-intensive operators into a single kernel. The number of operators fused in each pattern ranges from 2 to 48 and can achieve an average speedup of  $1.87\times$  (up to  $5.4\times$ ) compared to basic fusion methods such as those used in AnsoR. The most common pattern has been used 191 times in all models.

Such a general fusion capability often allows WELDER to outperform the model-specific implementations optimized by experts. For example, FasterTransformer [2] is a manually-optimized benchmark for transformer models from NVIDIA. It supports both element-wise fusion, such as BiasAdd+Transpose, and non-element-wise fusion, such as Layernorm+Softmax. In WELDER, all these patterns can be automatically fused. Even more, WELDER can further fuse Q\*K with the following Softmax in the attention block when the sequence length is not long (e.g., they are fused

Fused operators	# Ops
DepthwiseConv2dNative Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply Convolution Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply Convolution Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply Convolution Broadcast Add	48
Dot Relu Dot Relu Dot Relu Dot Relu Dot Relu Dot Relu Dot Relu Dot	13

Table 4: Examples of fusion patterns discovered by WELDER.

in BERT where the sequence length is 128, but are not fused in Conformer where the sequence length is 512, this is automatically decided by WELDER).

For the three models supported by FasterTransformer, we compare its performance with WELDER in Table 3. In general, WELDER achieves an average speedup of  $1.11\times$  (up to  $1.73\times$  on ViT) over FasterTransformer. Based on our profiled data, The notable speedup for ViT under batch size of 1 can be attributed to a convolution operator with a non-conventional shape, where both stride and kernel size are 32 (ViT’s patch size). For this single operator, WELDER’s generated kernel is 4.4x faster. This highlights WELDER’s adaptability in managing new operator shapes or model patterns.

Another example is NeRF, a popular 3D scene generation model that is typically implemented as a 7-layer MLP. To take full advantage of GPUs, domain experts often need to implement such models from scratch to achieve better fusion result (e.g., fully-fused MLP in [35]). With WELDER, we can automatically fuse this 7-layer MLP into a single GPU kernel. The generated kernel uses TensorCore for the first 6 layers and uses SIMT Core for the output layer, with all intermediate results stored in shared memory. We observe that our automatic fusion result can achieve a similar speedup (over  $5\times$ ) to the values reported in [35] (we are unable to evaluate their code [34] as it does not support V100 GPUs).

Finally, for CNN models such as NAFNet, BSRN, and MobileNet, WELDER is able to fuse different types of convolutions with other operators (e.g., Pooling, PixelShuffle, etc.). For example, in NAFNet, our system can fuse back-to-back pointwise convolutions together with the normalization

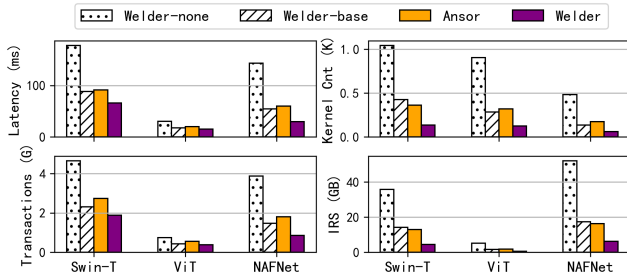


Figure 13: Latency, GPU kernel count, global memory transactions executed and intermediate result size (IRS) For 3 selected models (FP32, batch size 64).

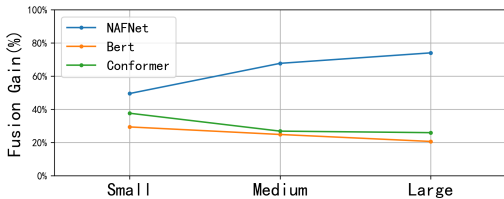


Figure 14: Varying input size, comparing with WELDER-base.

operations between them. Another interesting pattern is in models with multiple separable convolution layers, where each layer consists of two operations: a depthwise convolution (DWConv) and a pointwise convolution (PWConv). WELDER is able to determine the optimal fusing order for these two types of operators based on their operator configurations. For example, on the top layers where the feature maps are large and the number of channels is small, WELDER constructs a DWConv+PWConv fusion group because it is better to cache a complete feature map in shared memory. In contrast, on the bottom layers, WELDER constructs a PWConv+DWConv fusion group which caches a complete channel for DWConv to reuse, as the feature map becomes smaller.

**Ablation and sensitivity study.** To demonstrate the benefits of the holistic memory optimization provided by WELDER, we create two variants of WELDER: “WELDER-none” disables all inter-operator tile connection and only searches for intra-operator schedules, and “WELDER-base” only enables inter-operator tile connection at the register layer. We also include AnsoR in this experiment, as it is another codegen-based approach similar to ours. As shown in Figure 13, enabling register layer tile connection, WELDER-base reduces latency by an averaged 52% (i.e.,  $2.08\times$  speedup), kernel launch count by 67%, global memory transactions by 52% and intermediate result size (IRS) by 66% compared with WELDER-none. Note that the metrics of WELDER-base is similar to that of AnsoR, demonstrating the efficiency of our general tile-based memory scheduling compared with the rule-based fusion in AnsoR. Moreover, by enabling tile connection at shared memory layer, WELDER is able to further reduce latency by an averaged 29% (with up to  $1.82\times$  speedup), kernel launches by 60%, transactions by 25% and IRS by

Model	AnsoR time(s)	AnsoR Trials	WELDER Time(s)	WELDER Trials
BERT	15285	8000	244	651
Mobilenet	45527	25600	561	927

Table 5: Compilation time of AnsoR and WELDER

Model	AnsoR	WELDER	TensorRT
Resnet50	2.403	2.327	2.351
Resnet18	1.071	1.094	1.158
UNet	8.670	9.251	4.429
VGG16	4.267	4.123	2.584

Table 6: Performance on compute intensive models

65% compared with WELDER-base. Note that the reduction of memory transactions is less than the reduction of IRS, because memory access on the model weights part cannot be optimized by fusion.

In addition, we conducted a sensitivity study by varying the input sizes of three selected models: BERT (128-512 text length), Conformer (128-512 audio frames), and NAFNet (256x256-1024x1024 image input). The results, as depicted in Figure 14, reveal that the fusion gain significantly increases for NAFNet when employing larger images. Conversely, the gain diminishes for the other two transformer-based models. This discrepancy can be attributed to the fact that transformer-based models exhibit quadratic computational growth with respect to the input sequence length, thereby reducing their memory-intensive nature.

**Compilation time.** Table 5 compares WELDER’s compilation time against AnsoR, which is a search-based compiler requiring many tuning and profiling trails. We chose not to include other baselines in the comparison since they directly invoke library kernels, thereby eliminating the need for extra time dedicated to tuning and code generation. It shows that the end-to-end compilation speed of WELDER is more than an orders of magnitude faster than AnsoR. This is because AnsoR generates a very large search space for all the operators, and implicitly optimizes data reuse through machine learning-based tuning. This often requires a large number of tuning trials (e.g., 800 per operator in our evaluation) and has additional overheads to train a cost model on the fly. In contrast, WELDER decomposes the optimization space using a layered scheduling policy and searches for efficient tiling configurations using an analytic cost model to estimate traffic costs. As a result, WELDER requires significantly fewer tuning trials (20 per subgraph in our evaluation) than AnsoR.

**Performance on compute intensive models.** Traditional models like ResNet [21], VGG [43], and UNet [40] are typically dominated by some large operators such as convolution. For these compute intensive models, although WELDER mainly focuses on memory access optimization, WELDER can mostly achieve comparable performance to state-of-the-art baselines like TensorRT. This is because WELDER

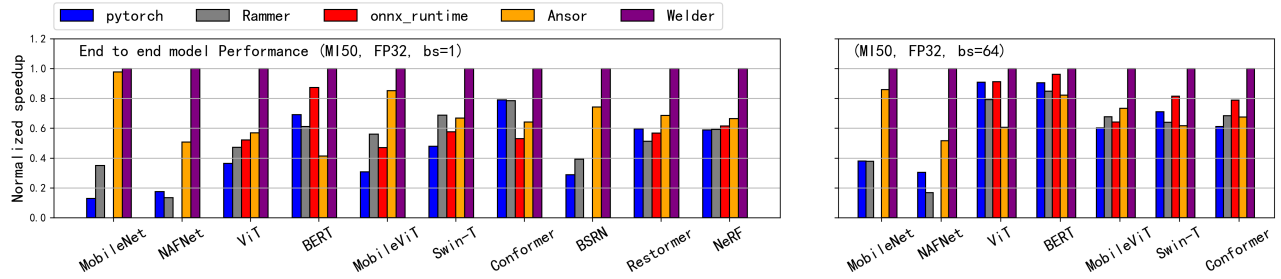


Figure 15: End-to-end model inference performance on AMD ROCm MI50 GPU (left : batch size of 1, right : batch size of 64).

can still generate high performance single operators (using the multi-level tiling abstraction, which is similar to Ansor [50] or Roller [52]) although there might be few chances to connect the tile at a higher memory level. However, for some convolution operators, existing libraries like cuDNN [4] implement them using an optimized numerical algorithm (e.g., winograd [26]), which are difficult to automatically derive from tensor expressions. This can result in WELDER performing worse than TensorRT if there is no additional memory optimization room to compensate for this gap. For example, Table 6 compares the performance of WELDER, Ansor, and TensorRT on four such models. For ResNet, both systems achieve comparable performance, as the majority of convolution operators in this model perform better when implemented with the DirectConv algorithm (which is supported by both Ansor and WELDER) instead of winograd. However, for UNet and VGG16, the dominant convolution operators are mostly implemented using winograd in TensorRT, and there are no further fusion opportunities for WELDER to exploit, resulting in better performance for TensorRT. Given that this is orthogonal to WELDER’s optimization, we leave the support of the winograd algorithm (by rewriting tensor expressions) to our future work.

### 5.3 Evaluation on AMD ROCm GPUs

We evaluate the efficiency of WELDER on AMD ROCm GPUs by comparing its performance with PyTorch, ONNXRuntime and Ansor. TensorRT and AStitch are not included because they are exclusive to NVIDIA GPUs. Figure 15 shows the end-to-end performance of the 10 DNN models. Compared with PyTorch, ONNXRuntime and Rammer, WELDER can outperform them by an average of  $2.62\times$ ,  $1.71\times$  and  $2.14\times$ , respectively. Compared to Ansor, WELDER achieves an average performance improvement of  $1.53\times$ . Figure 15 also shows the performance comparison with a larger batch size of 64, where WELDER outperforms PyTorch, ONNXRuntime, Rammer and Ansor by an average of  $1.69\times$ ,  $1.23\times$ ,  $1.86\times$  and  $1.47\times$ , respectively. Note that we have excluded some CNN models for ONNXRuntime as they fail to execute on it. We notice that WELDER’s speedup on MI50 is slightly smaller than that of V100, this is because MI50’s peak FLOPS is weaker than V100’s, while its peak bandwidth is higher,

Model	Image Size	Device	WELDERBase(s)	WELDER(s)
UNet	8k*8k	GPU	38.2	14.5
VGG16	8k*8k	GPU	15.7	8.30
UNet	2k*2k	IPU	31.1	8.56
VGG16	2k*2k	IPU	4.98	1.61

Table 7: Scale-up large DNN models to host memory

according to the official data-sheet. Such difference makes the workload more compute-intensive on MI50, leaving less optimization chances for memory access optimization.

### 5.4 Scale-up with Host Memory

WELDER’s abstracted device layer allows us to extend the memory hierarchy to support large DNN tasks. For example, in cases where classical CNN models like UNet or VGG16 are used to process high-resolution medical images [42], a single tensor from some layers is often too large to fit in the GPU memory. In these scenarios, tensor-based memory swapping optimization techniques, such as SwapAdvisor [22] or Capuchin [37], may not be effective due to the large tensor granularity. WELDER addresses this issue by generating a tile-based execution plan on the extended memory hierarchy through holistic traffic optimization. This approach allows us to load a data tile from the host memory, compute several connected operator tiles by reusing the data in device memory, and store the result back, as if it was being processed on a single device. To evaluate the efficiency of this scheduling approach, we compared WELDER with a variant that only disables data reuse at the device memory layer.

**Scale-up GPUs.** As a preliminary experiment, Table 7 shows the performance of WELDER when scaling up UNet and VGG16 on large image data by augmenting the GPU memory with a host memory layer. As the results show, by enabling tile-connection at the device memory layer, WELDER is able to achieve average speedups of  $2.63\times$  and  $1.89\times$  for the two models, respectively. It also reduces host memory transfer by  $3.11\times$  and  $2.90\times$ . Note that the ratios of reduced memory traffic are higher than the actual speedup, as we have implemented double buffering (along with pinned memory and CUDA streams) to overlap some memory transfer with computation.



**Scale-up GraphCore IPU.** We also perform a preliminary evaluation of WELDER’s ability to scale up on the Graphcore IPU [3], which is a DNN accelerator with a distinct architecture from NVIDIA and AMD GPUs. The IPU is equipped with massively parallel MIMD processors and a relatively small device memory (i.e., 300MB), which poses a challenge for it to handle even medium-sized tasks. We apply the same tile-based scheduling to the two models for the IPU and set the input image size to 2048\*2048 to adapt to the IPU’s memory capacity. The results in Table 7 show that WELDER’s optimization is able to achieve average speedups of  $3.63\times$  and  $3.09\times$  for the two models, respectively. This improvement ratio is higher than that of the GPU, which is mainly due to that we disable the double-buffer optimization for the IPU due to its limited memory.

## 6 Discussion

WELDER’s design and implementation mainly focuses on static models. For dynamic model execution, there are two practical ways to address this. First, the dynamic graph can be transformed into static sub-graphs through JIT compilation, such as PyTorch JIT compile, which has become a standard practice in PyTorch 2.0. Then, WELDER can concentrate on optimizing the static sub-graphs, which are typically the computationally dominant part. Second, even though tensor shapes may be dynamic, the internal tile in each operator can be statically determined. This presents an opportunity for WELDER to generate a static tile-level fusion plan but leave the number of parallel tasks determined by the input tensor shape.

## 7 Related Work

Compiler optimization like operator fusion is a widely-used technique in DNN computation to reduce kernel launch overhead and improve data locality in faster memory. Compilers such as TVM [15], Ansor [50], XLA [12], DNNfusion [36] all support operator fusion at register level. Other compilers try to further fuse operators at shared memory, relying on either fusion rules for a set of known operator types (e.g., AStitch [51], Apollo [49], DeepCuts [24]) or specific template for a few operator combinations (e.g., Bolt [47]). Specialized DNN runtimes such as TensorRT [7] and ONNXRuntime [8] have incorporated expert-designed fusion rules for some common patterns in popular models such as the transformer-based models. In contrast, WELDER works for general operators implemented in tensor expressions without the assumption on operator types and decides on the best fusion memory layer automatically. This is because an operator’s resource usage behavior (memory- or compute-intensive) often depends on its shape, and therefore the fusion decision.

Systems like Rammer [31], HFuse [27], Nimble [25], etc., exploit better hardware parallelism utilization and reduce kernel launches by either horizontal fusion or scheduling par-

allel tasks through multi-stream and CUDA graph. WELDER builds upon Rammer by further exploring a complementary optimization to these systems, i.e., holistic memory optimization with a vertical fusion, resulting in a further speedup for memory-intensive models.

Ansor [50] and Roller [52] are representative tensor compilers that are focusing on intra-operator optimization through either loop optimization or tiling optimization. Especially, Roller [52] and Triton [44] also utilize the concept of tile to optimize kernel performance (e.g., intra-operator data reuse). In contrast, WELDER complements them by optimizing for intra- and inter-operator memory access holistically. WELDER generalizes the *tile* concept in Roller into a *tile-graph* abstraction, exposes a holistic tile-level scheduling space, and proposes an efficient scheduling mechanism over the holistic space and the explicit memory hierarchy.

Some works optimize for a specific pattern regarding to a type of models with more aggressive operator fusions, such as fully-fused MLP for the NeRF model [35], manually fused kernels for CNN models [46], and attention fusion for transformer models [2, 18]. Our evaluation shows that WELDER can achieve most of these fusions automatically and even produce new fusion patterns to help further optimization.

Moreover, kernel fusion techniques have been used in traditional image processing [38, 39] or HPC [45] areas. These efforts usually leverage domain-specific fusion rules for their workload. WELDER focuses on DNN workload, while it is applicable for general operators represented by tensor expressions. It is also potentially helpful for workload that can be implemented in tensor expressions in other domains.

## 8 Conclusion

By observing that modern DNN models are becoming increasingly memory intensive, we introduced WELDER, a DNN compiler that optimizes the execution efficiency based on a new tile-graph abstraction. WELDER is able to holistically optimize efficient intra- and inter-operator data reuse across multi-level memory hierarchy. WELDER is the first to unify all common operator fusions into a single framework, allowing for the discovery of 89 uncommon fusion patterns, with the largest one fusing 48 operators into a single kernel. This generality enables WELDER to significantly outperform state-of-the-art baselines. More importantly, WELDER provides a systematic approach to take advantage of emerging trends in the memory hierarchy, such as larger and more connected on-chip memory, in the future AI accelerators.

## Acknowledgement

We thank anonymous reviewers and our shepherd, Prof. Byung-Gon Chun, for their extensive suggestions. This work was partially supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

## A Artifact Appendix

### Abstract

WELDER provides end-to-end DNN model compilation with its new tile-graph abstraction. This artifact reproduces the main results of the evaluation on NVIDIA V100 GPU.

### Scope

This artifact will validate the following claims:

- End-to-end model performances. By reproducing the experiments of Figure 9, Figure 10, Figure 11, Table 3 and Table 6.
- Motivation experiments in Figure 1 and Figure 2.
- Ablation study in Figure 13.
- Compilation time in Table 5.
- GPU stale out experiments in Table 7.

### Contents

This artifacts includes all the source code to implement WELDER. We provide a docker file to setup environments. For each figure and table mentioned above, we provide a script to reproduce its result. Since there are more than 50 model test cases to compile to fully reproduce the results, which will cost a long time (especially for the Ansor's baseline), we also provide pre-compiled logs and models for NVIDIA V100 GPU. Please refer to the README.md file in the repository for more details.

### Hosting

The artifact is hosted at github repository<sup>1</sup>. Please use git to clone the repository and checkout to the `osdi2023welder` branch.

### Requirements

This artifacts requires a NVIDIA V100 GPU with CUDA driver supporting CUDA runtime larger than 11.0.

---

<sup>1</sup><https://github.com/microsoft/nnfusion/tree/osdi2023welder>

## References

- [1] BladeDISC. <https://github.com/alibaba/BladeDISC>.
- [2] FasterTransformer. <https://github.com/NVIDIA/FasterTransformer>.
- [3] IPU PROGRAMMER'S GUIDE. <https://www.graphcore.ai/docs/ipu-programmers-guide>.
- [4] NVIDIA cuDNN. <https://developer.nvidia.com/cudnn>.
- [5] NVIDIA cutlass. <https://github.com/NVIDIA/cutlass>.
- [6] NVIDIA Tensor Cores. <https://www.nvidia.com/en-us/data-center/tensor-cores/>.
- [7] NVIDIA TensorRT. <https://developer.nvidia.com/tensorrt>.
- [8] ONNX Runtime. <https://github.com/microsoft/onnxruntime>.
- [9] onnxconverter\_common. <https://github.com/microsoft/onnxconverter-common>.
- [10] PyTorch. <https://pytorch.org/>.
- [11] TensorIR. <https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872>.
- [12] XLA. <https://www.tensorflow.org/xla>.
- [13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.
- [14] Liangyu Chen, Xiaojie Chu, Xiangyu Zhang, and Jian Sun. Simple baselines for image restoration. *arXiv preprint arXiv:2204.04676*, 2022.
- [15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.
- [16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
- [17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [18] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [19] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. *STOC '72*, page 143–150, New York, NY, USA, 1972. Association for Computing Machinery.
- [20] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.
- [21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [22] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.
- [23] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.
- [24] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile GPU workloads. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, pages 190–205. ACM, 2021.

- [25] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In *NeurIPS*, 2020.
- [26] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.
- [27] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27. IEEE, 2022.
- [28] Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76, 2016.
- [29] Zheyuan Li, Yingqi Liu, Xiangyu Chen, Haoming Cai, Jinjin Gu, Yu Qiao, and Chao Dong. Blueprint separable residual network for efficient image super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 833–843, June 2022.
- [30] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *CoRR*, abs/2103.14030, 2021.
- [31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.
- [32] Sachin Mehta and Mohammad Rastegari. Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer. *arXiv preprint arXiv:2110.02178*, 2021.
- [33] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.
- [34] Thomas Müller. *tiny-cuda-nn*, 4 2021.
- [35] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372*, 2021.
- [36] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 883–898. ACM, 2021.
- [37] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating (ASPLOS'20)*, 2020.
- [38] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *21st International Workshop on Software and Compilers for Embedded Systems, (SCOPE'S'18)*, pages 76–85. ACM, 2018.
- [39] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*, pages 242–253. IEEE, 2019.
- [40] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.
- [41] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.
- [42] Nahian Siddique, Sidike Paheding, Colin P. Elkin, and Vijay Devabhaktuni. U-net and its variants for medical image segmentation: A review of theory and applications. *IEEE Access*, 9:82031–82057, 2021.
- [43] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.
- [44] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.
- [45] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 191–202. IEEE Computer Society, 2014.



- [46] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. Accelerating deep learning inference with cross-layer data reuse on gpus. In *European Conference on Parallel Processing*, pages 219–233. Springer, 2020.
- [47] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.
- [48] Syed Waqas Zamir, Aditya Arora, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Ming-Hsuan Yang. Restormer: Efficient transformer for high-resolution image restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5728–5739, 2022.
- [49] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In *Proceedings of Machine Learning and Systems*, volume 4, pages 1–19, 2022.
- [50] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Anso: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.
- [51] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022*, page 359–373, New York, NY, USA, 2022. Association for Computing Machinery.
- [52] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

# Effectively Scheduling Computational Graphs of Deep Neural Networks toward Their Domain-Specific Accelerators

Jie Zhao

Information Engineering University

Siyuan Feng

Shanghai Jiao Tong University

Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyuan Lv, Qikai Xie  
Stream Computing Inc.

## Abstract

Fully exploiting the computing power of an accelerator specialized for deep neural networks (DNNs) calls for the synergy between network and hardware architectures, but existing approaches partition a computational graph of DNN into multiple sub-graphs by abstracting away hardware architecture and assign resources to each sub-graph, not only producing redundant off-core data movements but also under-utilizing the hardware resources of a domain-specific architecture (DSA).

This paper introduces a systematic approach for effectively scheduling DNN computational graphs on DSA platforms. By fully taking into account hardware architecture when partitioning a computational graph into coarse-grained sub-graphs, our work enables the synergy between network and hardware architectures, addressing several challenges of prior work: (1) it produces larger but fewer kernels, converting a large number of off-core data movements into on-core data exchanges; (2) it exploits the imbalanced memory usage distribution across DNN network architecture, better saturating the DSA memory hierarchy; (3) it enables across-layer instruction scheduling not studied before, further exploiting the parallelism across different specialized compute units.

Results of seven DNN inference models on a DSA platform show that our work outperforms TVM and AStitch by  $11.15\times$  and  $6.16\times$ , respectively, and obtains throughput competitive to the vendor-crafted implementation. A case study on GPU also demonstrates that generating kernels for our sub-graphs can surpass CUTLASS with and without convolution fusion by  $1.06\times$  and  $1.23\times$ , respectively.

## 1 Introduction and Background

Due to the slowing down of Moore’s Law, moving to DSAs is acknowledged as promising to meet the keen desire of DNNs for computing power [24]. After several years of investigation on accelerators specialized for DNNs [7, 8, 15, 22, 25, 29, 32, 55, 58], a commonly used DSA abstraction depicted on the left of Fig.1 has been formed for this application domain, based on which most existing DNN accelerators are manufactured.

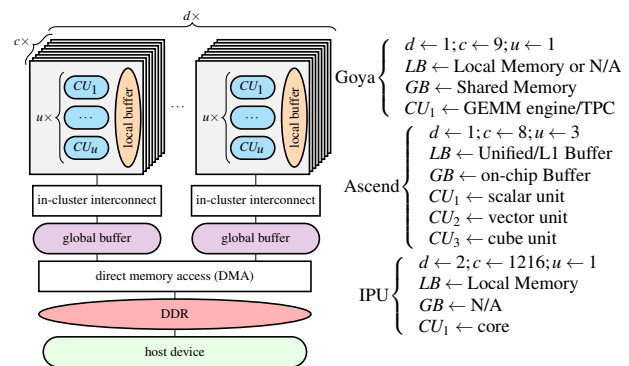


Figure 1: DNN DSA and its vendor customization.

We take the Habana Goya accelerator [32], the customized configuration of which is shown on the right of Fig.1, as an example to explain this abstraction. It is composed of  $d = 1$  cluster, each including  $c = 9$  cores that contains  $u = 1$  compute unit (CU) for different DNN tasks. CUs are either a tensor-processing core (TPC) with a scratchpad local buffer (LB) or a general matrix multiplication (GEMM) engine with no LB. Cores are connected using an in-cluster interconnect mechanism, equipped with a scratchpad global buffer (GB). Clusters, if  $d > 1$ , are stacked, communicating data with DDR via DMA. We also show the customized configurations of a Huawei Ascend 910 platform [29] and a Graphcore IPU device [22] in Fig.1. The Graphcore IPU uses the term “tile” to denote a core and its unique LB. Hardware architecture of others [7, 8, 15] can also be deduced according to the abstraction in Fig.1.

Hence, effectively scheduling DNNs toward this abstraction is essential to exploit the computing power of DNN accelerators for DSA compilers. Specialized for machining learning (ML) applications, these accelerators exhibit a scratchpad-based memory hierarchy and parallelism across both multiple cores and several CUs, but prior work [5, 13, 31, 54] devised to schedule DNNs on these DSA accelerators did not consider hardware architecture when partitioning a computational graph of DNN, introducing redundant off-core data move-

ments (*i.e.*, between LB and GB/DDR) and under-utilizing both faster local memory and parallelism across CUs.

## 1.1 Concepts and Notations

To explain the issues of prior work, we first introduce computational graphs, which are used by existing ML frameworks [1, 38] to represent DNN models. Fig. 2 is an example. As it can contain a large number of nodes, each of which performs a computation task on several tensors, a computational graph usually references memory footprints that exceed the local memory capacity of its target platform and thus cannot be scheduled as a whole. Existing schedulers first partition it into sub-graphs and next assign resources to each sub-graph. A sub-graph, which is also known as a fused operator (*op*), is first initialized by a graph node and next grouped according to its producer-consumer relations with others, implemented as a kernel function or kernel executable on target platform.

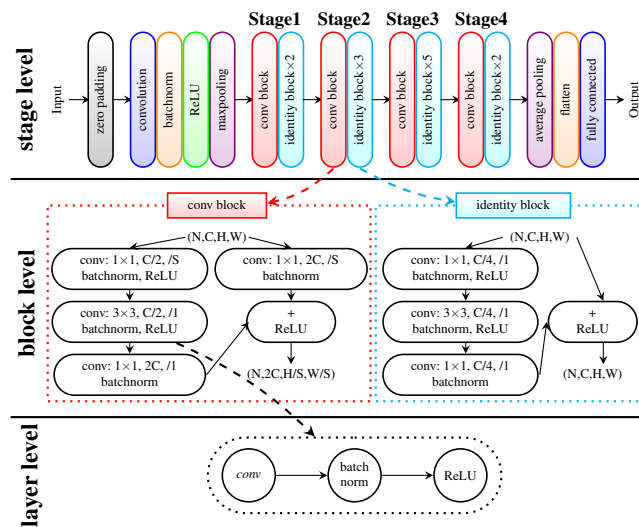


Figure 2: Computational graph of ResNet-50 [16]. A node (a circle) represents an *op*, and an edge (a solid arrow) denotes a producer-consumer relation of two *ops*. An *op* is a function that takes as inputs one to several tensors and outputs another. At the bottom is a  $3 \times 3$  convolution (*conv*) layer composed of three nodes. A dashed arrow connects a stage/block with its internal structure; a dotted box denotes a block. Other layers can be expressed in a similar way.

We use the term “layer” to denote a set of nodes connected in a straight-line manner, at most one out of which contains parameters that should be learned using the gradients of the loss. A graph node represents an *op*, which is traditionally referred to as a neural layer in neural networks. Some neural layers, however, do not require parameters to be learned (*e.g.*, ReLU) or learned without using the gradients of the loss (*e.g.*, batch normalization) during the training process, and they can thus be considered as the auxiliary *ops* of those that indeed

require parameters, *e.g.*, the convolution. We define layers as such because this definition summarizes the *op* fusion rules widely used by existing compilers [5, 53].

We also use the term “block” to represent an individual layer or a component composed of multiple layers that is recursively used in a DNN computational graph. For instance, the *conv* block composed of five layers is used once in each stage of Fig. 2, while the identity block is used multiple times within each stage.

The term “stage” is a logical, high-level abstraction used in the architecture of the ResNet-50 model, taking the results of its preceding stage as inputs and generating output tensors. It is used to simplify the design of the network architecture but usually not considered by optimizing compilers.

## 1.2 Challenges of Prior Work

By obscuring hardware architecture, prior work [5, 19, 54] constrains sub-graph grouping within a layer [39] (the bottom level of Fig. 2) and produces fine-grained sub-graphs. As each sub-graph is implemented by one kernel, prior work produces more kernels and **requires more off-core data movements between kernels**. Going one level upper in Fig. 2 can observe five and four layers in the *conv* and identity blocks, so the entire network may require hundreds to thousands of kernels and off-core data movements of the same order of magnitude [21], resulting in high pressure on the limited memory bandwidth of a DSA platform. In addition, managing such a large number of off-core data movements for DSA is non-trivial because, unlike a general-purpose architecture reinforced by its mature hardware mechanisms, the hierarchical scratchpad memory of the later is still controlled by hand or software [37].

Even though managing the data movements across a DSA’s memory hierarchy is possible, generating fine-grained sub-graphs still **misses the across-layer instruction scheduling opportunities**. Once formed, each sub-graph is lowered to a loop nest pipeline, to which memory optimizations and loop transformations like tiling and fusion are applied to better utilize hardware resources. While their different compositions constitute the search space that existing autotuners [2, 6, 26, 57] navigate to select the optimal scheduling, across-layer instruction scheduling opportunities, *e.g.*, overlapping the memory promotion statement of weights and the computation task of a  $3 \times 3$  *conv* layer with those of its preceding  $1 \times 1$  *conv* layer in Fig. 2, are not covered by such spaces.

Finally, since **the imbalanced memory usage distribution**, which refers to a phenomenon where memory usages vary across network architecture [30], **is not exposed/exploited**, the above scheduling paradigm also **under-utilizes the faster local memory of DSA**. On the top of Fig. 2, ResNet-50 is partitioned into four stages, each composed of one *conv* block and two or more identity blocks. A *conv* block converts its input with shape  $[N, C, H, W]$  into  $[N, 2C, H/S, W/S]$ , performing a down-sampling operation when  $S > 1$ , but an

identity block does not change its tensor shapes. The memory usage of stage1 is  $\frac{S^2}{2} \times$  larger than that of stage2, and this property also exists in stage3 and stage4. If the faster memory of the target DSA is saturated when executing stage1, it will be under-utilized when executing the remaining stages.

### 1.3 Our Solution and the Organization of the Paper

To address the aforementioned issues on DSA platforms, we introduce a novel scheduling approach in this paper. First, as redundant off-core data movements are caused by fine-grained sub-graphs produced by existing tools [5, 13, 19, 44], our approach has to construct coarser-grained sub-graphs that can generate larger kernels, which can change massive output tensors originally exchanged through GB/DDR of Fig.1 into intermediate tensors that can stay in LB of the later, thereby converting many off-core data movements between kernels into on-core data exchanges within kernels. Second, to enable across-layer instruction scheduling outside the search spaces of prior work [31, 57, 59], a sub-graph constructed by our work must be able to group layers or even blocks like those of Fig.2, thus better hiding memory latency and exploiting the parallelism across CUs. Finally, to saturate the faster local memory of a DSA platform in the presence of imbalanced memory usage distribution [30], our method should consider the internal relations between coarser-grained sub-graphs such that a better scheduling order can be obtained.

With these considerations in mind, we design and implement a novel scheduling approach—GraphTurbo. §2 exemplifies the core idea and presents the overview of GraphTurbo. §3 explains how GraphTurbo constructs, splits and orders coarse-grained sub-graphs, to the results of which §4 generates larger kernels. §5 reports the experimental results of seven DNN inference models on a DSA platform, which demonstrate that, while achieving performance close to the vendor-crafted implementation, GraphTurbo outperforms TVM [5] and AStitch [60] by  $11.15 \times$  and  $6.16 \times$ , respectively. A case study on GPU also shows that GraphTurbo can surpass CUTLASS [27] with and without *conv* fusion by  $1.06 \times$  and  $1.23 \times$ , respectively. Finally, §6 discusses the related work, and §7 concludes the work.

### 1.4 Contributions

In summary, this paper makes the following contributions.

- We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.
- This synergy reduces off-core data movements, better saturates the valuable local memory, and empowers across-layer instruction scheduling.

- We design and implement a novel scheduling approach GraphTurbo, addressing the deployment of DNNs on DSA chips and offering insight to other platforms.
- The experimental results demonstrate that GraphTurbo can outperform two state-of-the-art tools and achieve performance comparable to the vendor-crafted code.

## 2 Core Idea and Overview

This section first explains the core idea of GraphTurbo and next presents its overview.

### 2.1 Exemplifying the Core Idea

We use Fig.2 that classifies a batch of input images into different categories as an example to explain our core idea. Data parallelism is exploited across the  $d$  clusters of Fig.1, which is always possible due to the multi-dimensional parallelism of tensors. Decomposing the input tensors of a DNN model into  $d$  clusters can be achieved by splitting one or multiple parallelizable dimensions. We assume the batch dimension of size  $N = 32$  is split across these clusters, so each cluster processes  $n = 8$  images that have been offloaded to GB of Fig.1.

Our work studies how a DNN model is scheduled within one cluster. **The core idea is to maximally preserve the input tensors in LB in order to convert as many off-core data movements as possible into on-core data exchanges.** For the sake of clarity, we reproduce the stages of Fig.2 in Fig.3a and assume that each stage performs a down-sampling operation with  $S = 2$ .

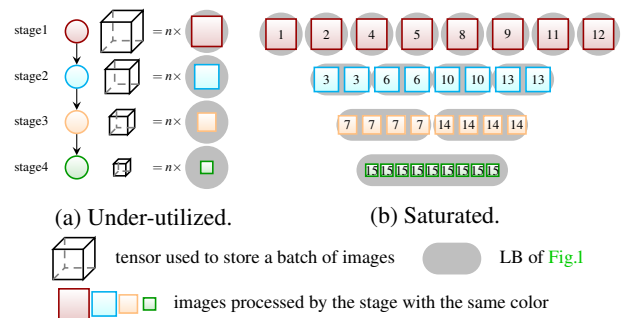


Figure 3: Utilization of LB under different scheduling methods. Timestamps in (b) define the scheduling of GraphTurbo.

Existing approaches [5, 13] can construct a sub-graph larger than a layer by grouping smaller ones, but they do not know when the grouping should terminate without hardware architectural information. Even though larger sub-graphs could be constructed for each stage of Fig.3a, these methods only schedule these sub-graphs according to their coarser-grained dependencies as the arrows show in Fig.3a, which produces a scheduling strategy that distributes each sub-graph of a stage onto the  $c$  cores of Fig.1. Suppose that the batch dimension is



split, then each core processes one image. If an image saturates LB when executing stage1, it will under-utilize this local memory when its core executes the later three stages, since their preceding stages reduce the tensor size by  $2\times$ ,  $4\times$  and  $8\times$ , respectively, by performing down-sampling operations.

GraphTurbo can easily construct large sub-graphs for each stage by synthesizing network and hardware architectures. It splits these large sub-graphs into eight, four, two, and one instance, respectively, converting the coarse-grained dependencies between large sub-graphs into fine-grained ones between sub-graph instances. By eliminating redundant fine-grained dependencies, GraphTurbo executes two instances of stage1’s sub-graph at timestamp 1 and 2 in Fig.3b, saturating LB while exploiting the parallelism across cores by distributing other parallelizable dimensions across them.

Next, GraphTurbo executes one instance of stage2’s sub-graph at timestamp 3, which processes two images, both in LB, as shown in Fig.3b, because the image size is decreased by  $2\times$ . LB is thus not under-utilized. The parallelism across  $c$  cores is exploited by distributing both the batch and other parallelizable dimensions. Readers can follow the timestamps to infer the scheduling order and find that LB is never under-utilized while fully exploiting the parallelism between cores. In particular, this scheduling approach achieves a  $7.97\times$  speedup over TVM on our experimental platform.

## 2.2 Overview of GraphTurbo

To obtain scheduling strategies similar to Fig. 3b, GraphTurbo takes in a computational graph simplified by some standard graph optimizations [13,23] and schedules it at graph level (§3). To construct coarser-grained sub-graphs, e.g., for stages of Fig.3a, GraphTurbo first collects hardware information (§3.1) to guide its grouping process (§3.2) by synthesizing network and hardware architectures. These sub-graphs are then split into instances, which are sorted (§3.3) to achieve the scheduling order like in Fig.3b. How parallelism is exploited and load balance is guaranteed across multiple cores are then explained (§3.4), with core binding and memory scopes automatically inferred. A concatenation step is then used to collect the tensors of producer sub-graph instances (§3.5), followed by some generalization discussions (§3.6).

The graph scheduler produces ordered sub-graph instances, which are delivered to the kernel generator (§4), producing kernels by combining loop fusion (§4.1) and buffer stitching (§4.2), with memory allocation and reuse automatically managed (§4.3). For the example in §2.1, the graph scheduler concentrates on input images. The convolution weights of this model are getting larger but only used within layers and do not result in communications between stages. GraphTurbo only allocates a small, fixed size of buffers in LB to allow for the promotion of such tensors to local memory when handling each layer, and the overhead of such memory promotion statements is hidden behind the computation (§4.4).

## 3 Scheduling Sub-graph Instances

This section constructs coarser-grained sub-graphs and schedules their instances. To achieve this goal, we need to address five issues and thus organize this section into five steps, with the difficulties explained at the beginning of each.

### 3.1 Collecting Splitting Information

GraphTurbo relies on producer-consumer relations between sub-graphs to group them into larger ones. This strategy, however, does not know when to stop without knowing hardware architectural information. Hence, **this section first collects hardware knowledge for sub-graphs**. As it will also be used to split sub-graphs (§3.3), we refer to it as splitting information `SplitInfo`, which is a set of 4D tuples ( $split_d, n_d, f_d, d$ ). Algo.1 summarizes how to compute it.

---

#### Algorithm 1: Compute SplitInfo

---

```

1 SplitInfo  $\leftarrow \emptyset$ ;
2 foreach  $d$  in  $[1, \dots, depth \leftarrow \text{dimof}(\text{output of } SG)]$  do
3    $n_d \leftarrow 0$ ;  $split_d \leftarrow 0$ ;  $f_d \leftarrow \infty$ ;
4   foreach  $v$  in  $[1, 2, 4, 8, 9, \dots, size_{output}^{(d)}]$  do
5     if  $\lceil \frac{peak}{v} \rceil \leq \text{sizeof}(\text{LB})$  then
6        $n_d \leftarrow n_d + 1$ ;  $split_d \leftarrow 1$ ;  $f_d \leftarrow v$ ; break;
7   foreach  $t$  in intermediates do
8     if  $split_d = 1$  then
9        $n_d \leftarrow n_d + \text{num\_of\_op}(t)$ ;
10      if match_dim( $t, d$ ) and  $size_t^{(d)} \% f_d = 0$  then
11        SplitInfo  $\leftarrow$  SplitInfo  $\cup \{(split_d, n_d, f_d, d)\}$ ;

```

---

Before grouped, a sub-graph  $SG$  is a node that represents an *op*. When lowered, it may produce several loop nests since the *op* it represents can be complex such that multiple intermediate tensors are used to realize its function [53]. Except the last one that defines the output tensor, all remaining loop nests write to intermediate tensors. Algo.1 computes `SplitInfo` by first splitting the output tensor (lines 3-6) and next propagating its splitting information to each of intermediate tensors (lines 8-11) due to the following reasons.

First, a sub-graph has one output but its input tensors can be many. Considering only the output tensor simplifies the algorithmic design. Second, it is the output that determines how the loop nests of this sub-graph should be split or tiled [41,52]. Once the information of the output and intermediate tensors is determined, how input tensors should be split is also known.

Indeed, computing `SplitInfo` this way may introduce recomputation of intermediate or input tensors, which would be expensive when fusing multiple *conv* or matrix multiplication *ops*. We thus use a simple cost model to prevent excessive recomputations that offset the benefits brought by fusion.

$depth$  represents the loop nest depth of the output tensor. By iterating a loop dimension  $d$  from the outermost to inner (line 2), Algo.1 makes use of the parallelism across cores as early as possible. Next, Algo.1 defines three metrics (line 3),

$split_d, f_d, n_d$ , that represent whether the current dimension  $d$  can be split, the splitting factor of this dimension, and the number of *ops* split by it, respectively.

$v$  iterates the values of line 4 to instantiate  $f_d$ . We consider  $size_{output}^{(d)}$  that denotes the loop extent of the current dimension  $d$  as the upper bound because  $v > size_{output}^{(d)}$  does not split the current dimension  $d$ . The first three values decompose the dimension  $d$  into eight, four, and two cores, while guaranteeing load balance across them. The first three stages in Fig.3b are split this way. Values between  $8 \leq v \leq size_{output}^{(d)}$  do not exploit the parallelism of the current dimension  $d$  across cores but parallelize other dimensions, with load balance across cores fully considered. The splitting of stage1 in Fig.3b is an example of this case. A value is used to instantiate  $f_d$  (line 6) if the size of memory footprints,  $\lceil \frac{peak}{v} \rceil$ , required by a sub-graph instance does not exceed the memory capacity of LB (line 5).  $peak$  is the size of memory footprints required by  $SG$ .  $n_d$  and  $split_d$  are also updated accordingly. As a smaller  $v$  partitions  $peak$  into larger pieces, the  $v$  loop here is a greedy strategy.

$t$  iterates each intermediate tensor (line 7). It takes in the dimension  $d$  and first inspects whether the dimension can split the output tensor (line 8).  $n_d$  is increased by the number of *ops* in  $t$  (line 9) if this condition is satisfied. In addition, the 4D tuple is added to  $SplitInfo$  (line 11) if the dimension  $d$  matches one loop dimension of  $t$  and the loop extent of the matched dimension  $size_t^{(d)}$  is dividable by  $f_d$  (line 10).

By exactly computing  $SplitInfo$  for sub-graphs, GraphTurbo determines appropriate sizes for its generated kernels. As  $SplitInfo$  usually has several elements and each one encodes a loop dimension that can be split, GraphTurbo needs to select the best dimension for a sub-graph. We consider the following criteria for this issue. First, a loop dimension is better if it splits more *ops*. Second, a loop dimension with a smaller splitting factor is preferred since it tends to better saturate LB. Finally, a dimension with a smaller loop depth is considered superior since it exhibits outer parallelism and fewer communications. These criteria are modeled as computing the lexicographical maximum of an optimization problem

$$\text{lexmax}_{v,d \in SplitInfo} (n_d, -f_d, -d) \quad (1)$$

where the order of the three metrics defines the priorities.

### 3.2 Grouping Sub-graphs

Now we can group sub-graphs. GraphTurbo still performs this step according to producer-consumer relations, but it **constructs larger sub-graphs by leveraging  $SplitInfo$  to determine the termination of grouping**, which is not restricted within layers [5, 19, 54]. Algo.2 outlines the process.

Algo.2 first sorts a computational graph  $G$  by topologically ordering all of its  $g$  nodes (line 1), each of which is treated as one sub-graph  $SG$  and delivered to Algo.1 to compute its

#### Algorithm 2: Group sub-graphs

```

1  $SG[1, \dots, g] \leftarrow \text{topological\_order}(G); b \leftarrow g;$ 
2 foreach  $i$  in  $[1, \dots, g]$  do
3    $SplitInfo[i] \leftarrow \text{Algo.1}(SG[i]);$ 
4    $BestSplit[i] \leftarrow \text{Eq. (1)}(SG[i], SplitInfo[i]);$ 
5 repeat
6    $\{G, s\} \leftarrow \text{straight\_merge}(SG[1, \dots, b], SplitInfo[1, \dots, b]);$ 
7   foreach  $i$  in  $[1, \dots, s]$  do
8      $BestSplit[i] \leftarrow \text{Eq. (1)}(SG[i], SplitInfo[i]);$ 
9      $\{G, d\} \leftarrow \text{diamond\_merge}(SG[1, \dots, s], SplitInfo[1, \dots, s]);$ 
10    foreach  $i$  in  $[1, \dots, d]$  do
11       $BestSplit[i] \leftarrow \text{Eq. (1)}(SG[i], SplitInfo[i]);$ 
12       $\{G, b\} \leftarrow \text{branch\_merge}(SG[1, \dots, d], SplitInfo[1, \dots, d]);$ 
13      foreach  $i$  in  $[1, \dots, b]$  do
14         $BestSplit[i] \leftarrow \text{Eq. (1)}(SG[i], SplitInfo[i]);$ 
15 until  $s, d$  and  $b$  all do not decrease;
```

$SplitInfo$  (lines 2-3). Next, Algo.2 considers three different merging patterns (lines 5-15) to group these sub-graphs, which reduces the number of sub-graphs from  $g$  to  $s, d$ , and  $b$ , respectively. The sub-graph index and  $BestSplit$  are updated each time a merging pattern is grouped.

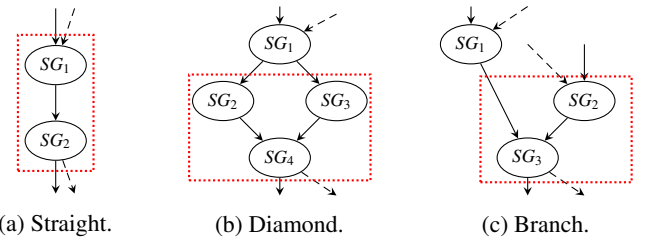


Figure 4: Merging patterns considered by GraphTurbo. A solid arrow is the producer-consumer relation, and a dashed one denotes a possible connection with another sub-graph.

The merging patterns considered by Algo.2 are defined as follows. First, two sub-graphs  $SG_1$  and  $SG_2$  form a straight pattern (Fig.4a) if and only if (iff)  $SG_1$  is the unique producer of  $SG_2$  and  $SG_2$  is the unique consumer of  $SG_1$ . Second, four sub-graphs make a diamond pattern (Fig.4b) iff there are only one entry, one exit and at least two paths from the entry to the exit. The entry ( $SG_1$ )/exit ( $SG_4$ ) can be a consumer/producer of multiple outside sub-graphs. Third, three sub-graphs constitute a branch pattern (Fig.4c) iff there exists only one exit and multiple paths to it, and  $SG_1$  could be but not necessarily a producer of  $SG_2$ .

Instead of grouping all components of a merging pattern, Algo.2 fuses a subset of them into a larger sub-graph  $SG$ , as shown by a red dotted box in Fig.4. Algo.2 uses two heuristic rules to determine whether the grouping is allowed.

- (i)  $SG_1$  and  $SG_2$  of Fig.4a (or  $SG_2$  and  $SG_3$  of Fig.4c) can be merged if two preconditions are satisfied. First, the splitting factor of  $SG_2$  of Fig.4a (or  $SG_3$  of Fig.4c) is no less than that of  $SG_1$  (or  $SG_2$ ). As a larger splitting factor partitions  $peak$  into smaller pieces, this precondition

prohibits the propagation of a sub-graph’s large splitting factor to its followers in the case of down-sampling operations. Second, the splitting factor of  $SG$  is not larger than that of  $SG_2$ , ensuring that the grouping result does not deteriorate the utilization of LB exploited by  $SG_2$ . `SplitInfo` of  $SG$  can be computed using [Algo.1](#).

- (ii)  $SG_2$ ,  $SG_3$  and  $SG_4$  of [Fig.4b](#) can be merged into  $SG$  if the splitting factor of  $SG$  is not larger than the maximum among the splitting factors of  $SG_2$ ,  $SG_3$  and  $SG_4$ , which is also used to guarantee good LB utilization.

We now explain how the *conv* and identity blocks of [Fig.2](#) are grouped. First, [Algo.2](#) identifies the straight patterns in each layer and obtains [Fig.5a](#). Specifically, the three layers on the left of the *conv* block in [Fig.2](#) is identified as a straight pattern, fused into one sub-graph denoted using label 1. Neither of the *conv* and the ReLU layers of this *conv* block is identified as a straight pattern; instead, they both form individual sub-graphs, represented using labels 2 and 3, respectively. Similarly, the three *conv* layers of the identity block constitute a straight pattern, depicted using the sub-graph with label 4; and its ReLU layer is denoted using label 5.

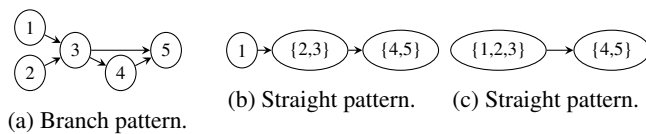


Figure 5: Merging the *conv* and identity blocks of [Fig.2](#).

Next, [Algo.2](#) finds two branch patterns (*i.e.*, sub-graphs 3 and 5 in [Fig.5a](#)) and produces [Fig.5b](#), the straight patterns of which are first merged into [Fig.5c](#) and finally form a single sub-graph  $\{1, 2, 3, 4, 5\}$ . The preconditions of the above two rules are all satisfied when merging these patterns. In practice, [Algo.2](#) can also group the multiple identity blocks and a *conv* block into a single sub-graph, thus producing four large sub-graphs for stages in [Fig.3a](#). These large sub-graph are no longer grouped because the second precondition of [Rule \(i\)](#) is not satisfied.

### 3.3 Ordering Sub-graph Instances

The synergy between network and hardware architectures enabled by [§3.1](#) and [§3.2](#) partitions a computational graph into larger sub-graphs. In addition, GraphTurbo also uses `SplitInfo` to split each sub-graph into instances, **the order of which is determined in this section**.

For instance, each stage in [Fig.3a](#) is converted into one to multiple labeled sub-graph instances with the same color, forming the new graph in [Fig.6](#). All instances at the same horizontal level are homogeneous and thus can be executed in any order. The edges between these instances are inherited from sub-graphs, but the gray ones are redundant and easily

eliminated. Specifically, we determine whether an edge is redundant or not by checking whether the output of a producer instance is used by one consumer instance. This is achievable because combining `SplitInfo` and the shape of an output tensor can perform this checking. The considered edge is redundant and eliminated if the checking result is true.

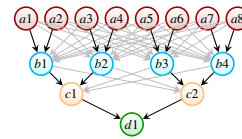


Figure 6: Sub-graph instances of the four stages in [Fig.3a](#).

GraphTurbo currently uses two approaches to schedule a computational group  $G$  of sub-graph instances. First, it visits sub-graph instances in a breadth-first search (BFS), producing a schedule order shown in [Fig.7a](#). Second, GraphTurbo visits them in a depth-first search (DFS), as described in [Algo.3](#). We currently only use the BFS and DFS searches because they simplify the algorithmic design of GraphTurbo. As will be reported in [§5](#), this choice achieves promising results. We are currently working on an integer linear programming approach to find a better solution than both of these search heuristics, which will be released soon.

[Algo.3](#) first instantiates a list *visit* by visiting  $G$  in any orders (line 1) and next moves all of its sub-graph instances with no in-degrees from *visit* to another list *ready* in order (lines 3-4). The in- and out-degrees of  $SGI$  are only determined by the black edges in [Fig.6](#). The last  $SGI$  with no in-degrees of *visit* (lines 6-8) is extracted and added into the ordered list *order* (line 9), with its consumers updated (lines 12-13) and moved from *visit* to *ready* (line 14).

---

#### Algorithm 3: Schedule Sub-graph Instances

---

```

1 visit ← get_subgraph_instances(G);
2 while visit ≠ ∅ do
3   foreach indegree(SGI) = 0 in visit do
4     ready ← ready.push(SGI); visit ← visit \ SGI;
5   while ready ≠ ∅ do
6     p ← sizeof(ready);
7     while indegree(SGIp) ≠ 0 do
8       p ← p - 1;
9     order ← order.push(SGIp); ready ← ready \ SGIp;
10    foreach SGI in visit and ready do
11      if SGIp is a producer of SGI then
12        remove_producer(SGI, SGIp);
13        indegree(SGI) ← indegree(SGI) - 1;
14        ready ← ready.push(SGI); visit ← visit \ SGI;
```

---

As an example, [Fig.7](#) illustrates how the three lists change when [Algo.3](#) is applied. In particular, as  $G$  can be visited in any order, we assume that it is visited in a BFS order for illustrative purpose. *order* is inspected from left to right. GraphTurbo finally selects the better one ([Fig.3b](#)) between the two results with respect to memory utilization. The labels

of each circle in Fig.7f would change if  $G$  is visited in other orders but the colors not, which does not matter because all instances of the same sub-graph are homogeneous.

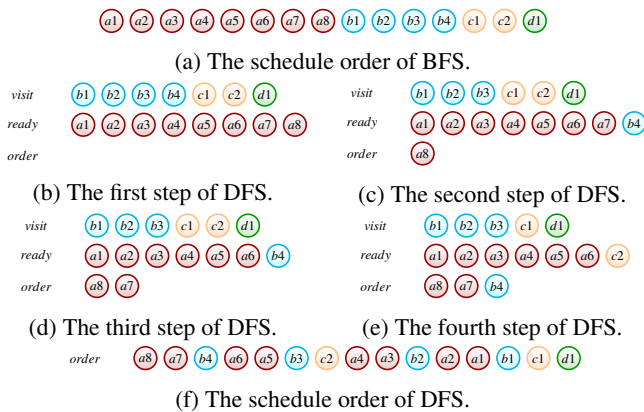


Figure 7: Different schedule orders of Fig.6. We only show the first four steps of Algo.3 for the sake of clarity.

### 3.4 Inferring Core Binding and Buffer Scopes

The next step is to **bind each ordered sub-graph instance to multiple cores**. A sub-graph instance’s binding strategy can be determined by inspecting the loop dimensions of its output tensor. However, determining binding strategies individually may introduce more communications due to the mismatching between them. We use Algo.4 to infer binding strategies.

As Algo.4 can detect the mismatching between a pair of producer-consumer sub-graph instances, it also uses information to **infer at which buffer scopes the output tensor of a sub-graph instance should be declared**. It first visits a schedule order  $O$  like Fig.7f in a reverse order and records all corresponding sub-graph instances in a list  $visit$  (line 1). Defined by default as an empty tuple  $\square$  and LB,  $bind[i]$  and  $scope[i]$  are used to record the binding strategy of a sub-graph instance and at which buffer level its output tensor is declared, respectively (line 2). As an output tensor is taken in by another sub-graph instance, we do not care about where the input tensors of a sub-graph instance should be declared. A sub-graph instance also produces intermediate tensors not considered by Algo.4, which GraphTurbo manages using its kernel generator (see §4.1).

Algo.4 infers binding strategies and buffer scopes for each sub-graph instance denoted by  $visit[i]$  (lines 3-16). If  $bind[i]$  is empty (*i.e.*, no information can be used for inference) or  $scope[i]$  is not LB (*i.e.*, the known information cannot be used for inference) (line 4), Algo.4 instantiates  $bind[i]$  using a plain strategy (line 5) and infers the binding strategy of the input tensors of the current sub-graph instance. A plain binding strategy is obtained by greedily allocating more cores from the outermost to inner loop dimensions of a tensor. The binding factors along multiple dimensions form a multi-dimensional

#### Algorithm 4: Infer Core Binding and Buffer Scopes

```

1  $visit \leftarrow \text{DFS\_visit\_reverse\_order}(O)$ ;  $size \leftarrow \text{sizeof}(visit)$ ;
2  $bind[1, \dots, size] \leftarrow \{\square\}$ ;  $scope[1, \dots, size] \leftarrow \{LB\}$ ;
3 foreach  $i$  in  $[1, \dots, size]$  do
4   if  $bind[i] = \square$  or  $scope[i] \neq LB$  then
5      $bind[i] \leftarrow \text{plain\_binding}$  (output of  $visit[i]$ );
6     if  $\text{infer\_binding}(bind[i]) = \square$  or is invalid then
7       continue;
8     foreach  $producer[j]$  in  $visit$  do
9       if  $bind[j] = \square$  then
10         $bind[j] \leftarrow \text{infer\_binding}(bind[i])$ ;
11       else if  $bind[j] \neq \text{infer\_binding}(bind[i])$  then
12         $scope[j] \leftarrow GB$ ;
13       else
14        continue;
15   else
16      $bind[i] \leftarrow \text{update\_binding}(bind[i])$  uses more cores than
        $\text{plain\_binding}$  (output of  $visit[i]$ ) ?  $\text{update\_binding}$ 
       ( $bind[i]$ ) :  $\text{plain\_binding}$  (output of  $visit[i]$ );

```

tuple. Algo.4 tries to instantiate a binding factor by iterating integers 8, 4, 2, and 1, which guarantees load balance across cores. The iteration turns into its next value if a loop extent  $size^d$  is not dividable by current one. Note that some tensors can have some specific requirements annotated to their loop dimensions, which cannot be bound to cores.

Next, Algo.4 uses  $\text{infer\_binding}$  to infer the binding strategies of  $visit[i]$ ’s input tensors, which are the output tensors of  $visit[i]$ ’s producers (line 8). The inference is achieved by first matching loop dimensions of an input tensor as line 10 of Algo.1 does and next propagating the binding factors of the matched dimension from the output tensor. A binding strategy can be invalid if it does not satisfy the annotated requirements. Hence, Algo.4 falls into one of the following three cases when the inferred binding strategy is neither empty nor invalid: (1)  $bind[j]$  is instantiated by the inferred binding strategy if it has not yet been defined (lines 9-10); (2)  $scope[j]$  is overwritten by GB if the already defined  $bind[j]$  does not match the inferred binding strategy (lines 11-12), since a communication is required here (see §3.5); (3) no mismatching between the already defined  $bind[j]$  and the inference succeeds, and Algo.4 steps into next iteration (lines 13-14).

$bind[i]$  is inferred and not empty in the **else** case (line 15), for which Algo.4 tries to update  $bind[i]$  by reconsidering the possibly annotated requirements of  $visit[i]$ ’s output tensor and computes a plain binding strategy for it. The one using more cores between these two binding strategies is finally used to rewrite  $bind[i]$  (line 16).

### 3.5 Concatenating Instance Outputs

As GraphTurbo splits a sub-graph into multiple instances, the output tensor of a sub-graph is also partitioned into multiple pieces. Hence, **this section introduces the step to concatenate the outputs of these sub-graph instances**.

To implement this step, GraphTurbo detects fine-grained



dependencies between sub-graph instances and introduces concatenation *ops* before each consumer of multiple producers, obtaining Fig.8 for the running example. A concatenation *op* is lightweight since its inputs and output stay in either LB or GB. GraphTurbo needs to insert additional *ops* for moving data across the memory hierarchy if the binding strategies and memory scopes of the tensors taken in by a concatenation *op* are different from each other and/or those of its output.

Fig.9 depicts two scenarios where such (gray) *ops* should be inserted. A sub-graph instance or an auxiliary *op* is denoted using an ellipse that displays the shape, scope and binding strategy of its output tensor. On the left, a copying *op* is introduced once mismatching between the memory scopes is captured, promoting data from its input (GB) to its output (LB). On the right, a redistribution *op* is inserted due to the difference between binding strategies, triggering a communication between cores. This can be achieved by resetting the tuple using the smaller binding factors along each dimension.

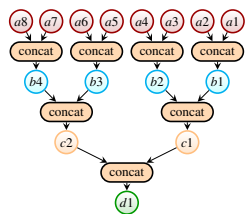


Figure 8: Concatenate instance outputs.

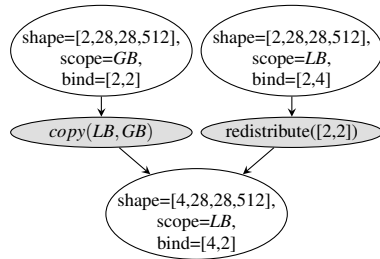


Figure 9: Insert data movement *ops*.

### 3.6 Generalizing the Approach

Now GraphTurbo can effectively schedule a DNN computational graph, but we made some assumptions in its design. This section discusses how they can be generalized. First, §3.1 assumes that a sub-graph has one output tensor, which is often the case in practice. One can repeat Algo.1 for each output tensor of a sub-graph that with multiple output tensors.

Second, the rules defined in §3.2 take into account down-sampling operations in a DNN model. However, our scheduler can also be generalized to target up-sampling operations by simply modifying Rule (i). A further split *op*, which can be considered as the opposite of the concatenation *op* introduced in §3.5 may be required to distribute the output of a sub-graph instance to its multiple consumers. We did not experiment with up-sampling operations in this work.

Third, §3.3 uses two methods to order sub-graph instances, which are simple but effective, as will be demonstrated in §5. We are now investigating another heuristic by allocating higher priorities to sub-graph instances with heavier memory footprints and plan to release it in the future.

Finally, GraphTurbo greedily uses LB, but, as a suggestion, it could be replaced by GB to schedule a computational graph

across the  $d$  clusters of Fig.1. It could also be substituted by faster memory of other platforms, e.g., shared memory of GPU. Interestingly, much larger LB sizes would simplify the algorithmic flow of GraphTurbo. Even when splitting a sub-graph instance is unnecessary, GraphTurbo could obtain a similar scheduling to TVM but with across-layer memory optimizations (§4.4) fully considered. Moreover, making use of the higher-level buffer, e.g., those residing in CUs of Fig.1 if any, is profitable, since data exchanges between such buffers and LB may dominate the communications in such cases.

## 4 Kernel Generation for Sub-graph Instances

Once scheduled sub-graph instances are obtained, the kernel generator can take each of them as input and lower them into loop nest pipelines. **The task of our kernel generator is to generate larger kernels by implementing loop transformations and stitching the intermediate tensor in the faster local memory of a DSA platform.** To minimize the engineering cost, we prototype our kernel generator in TVM [5], but it may fail to produce a single kernel for one sub-graph instance. Fig.10 exemplifies this issue by gradually expanding one sub-graph instance,  $b_3$ , of Fig.8.

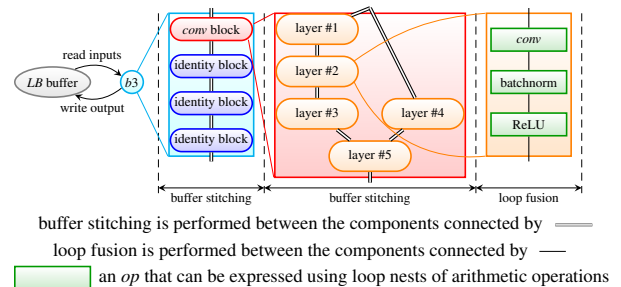


Figure 10: Expand  $b_3$  to generate a single kernel for it.  $b_3$  is sub-graph instance of stage2 of Fig.2. It is expanded to one *conv* block and three identity blocks, each of which is then expanded to multiple layers. How the *conv* block is expanded is shown in the middle, which obtains five layers shown on the left of the block level of Fig.2. These layers are labeled, and how layer #2 is expanded is shown on the rightmost, which produces the three *ops* shown at the layer level of Fig.2.

The rightmost part is what TVM’s kernel generator takes in, but Fig.10 shows that  $b_3$  is composed of 38 *ops* (11 for the *conv* block and 9 for each identity block), out of which 13 are *conv ops* (4 for the *conv* block and 3 for each identity block). Performing loop fusion across multiple *conv ops* is outside the power of TVM’s kernel generator. Although CUTLASS [27] was recently integrated into TVM to alleviate this problem for GPU [50], the number of acceptable *conv ops* is still limited. Furthermore, even if a similar vendor-crafted library can be offered on a DSA platform, the kernel generator would still put the output of a sub-graph instance in DDR, which in turn

would regress the benefit created by GraphTurbo.

## 4.1 Loop Fusion within Layers

The kernel generator can easily fuse *ops* within a layer. We use layer #2 in Fig.10 as an example and lower it to the loop nest pipeline shown in the middle of Fig.11. Since TVM requires users to write schedule templates for these loop nests, simply adopting its workflow cannot fully automate GraphTurbo.

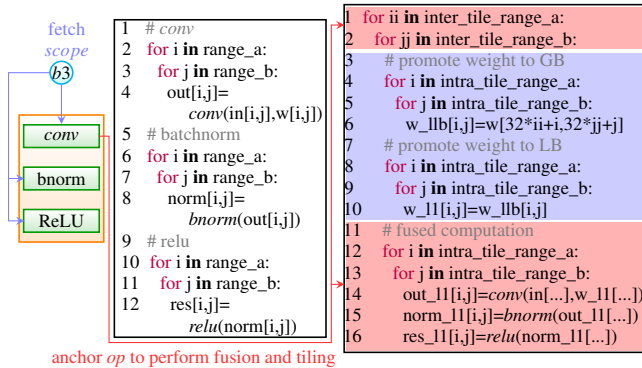


Figure 11: Loop fusion within a layer. Left: fetch the *scope* information of the sub-graph instance *b3* in Fig.8. Middle: the loop nest pipeline of layer #2 in Fig.10. Right: the tiled and fused loop nest, with memory promotion statements of weights automatically inserted. Red arrows connect the anchor *op* with its tiled loop dimensions. Blue arrows represent that the *ops* fetch the *scope* information from Relay IR.

To resolve this issue, GraphTurbo selects an anchor *op* out of each layer and automatically performs loop tiling on this *op*. An anchor *op* should be set using either *conv*/matrix multiplication if the later appears in the current layer, or the last *op* of the layer otherwise. In the later case, the anchor *op* should be an elementwise *op*. The outermost two loop dimensions are selected for tiling because they are parallelizable in both cases. The tile sizes along the two dimensions of an anchor *op* are then selected in a similar way to that used to determine a plain binding strategy in §3.4, which greedily maximizes the memory utilization of LB. In other words, a tile size is instantiated using a largest integer that not only divides the current loop extent but also allows the resulted tiled tensors to stay in LB. Once the tile shape and sizes of the anchor *op* are determined, the loop bounds of other *ops* can be inferred and fused with the anchor *op*, just like what existing techniques [41, 52] did, producing the fused and tiled loop nest shown in the red regions of Fig.11.

By converting the tensors written by the *conv* and *batchnorm* *ops* into intermediate ones, this method automatically allocates them in faster memory, as mentioned in §3.4. Before doing so, the kernel generator fetches the *scope* information of the current sub-graph instance, *i.e.*, *b3* in Fig.11, allocating intermediate tensors at the defined memory level. Memory

promotion statements of a weight tensor are also automatically injected in the same way, as shown by the blue region of Fig.11. Note that some *ops* like batch normalization can be folded, but we keep it here for illustrative purpose.

## 4.2 Buffer Stitching across Layers/Blocks

After the internal of a layer is fused, we do not put its output back to DDR but still let it remain in LB, *e.g.*, *res\_11* in Fig.11. Hence, all layers of the *conv* block can exchange their data via LB, which we refer to as *buffer stitching* and the kernels used to implement these layers can be wrapped into one. The input tensors of the *conv* block are also put in LB, as declared by *scope* in §3.4. An identity block can be handled in a similar way. As the output tensors of each block’s last layer also stay in LB, the four blocks can all be stitched together.

By targeting memory-intensive *ops*, AStitch [60] also implements a similar functionality. However, our work also considers compute-intensive *ops*. In addition, we also try to maximize faster local memory between sub-graph instances. By combining loop fusion and buffer stitching, our implementation generates a single kernel for the sub-graph instance *b3*. In contrast, TVM produces one kernel for each layer, increasing the number of generated kernels (65 in total) and thus requiring more off-core data movements.

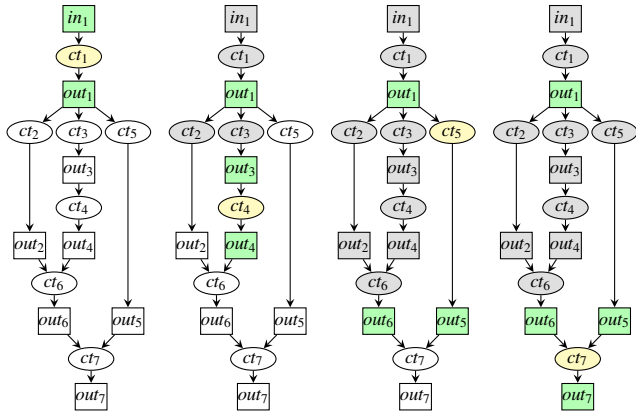
## 4.3 Memory Allocation and Reuse

The remaining task is to allocate space at the memory levels defined by *scope* for each tensor, which is trivial to implement. However, by putting many tensors in the faster local memory, GraphTurbo calls for a careful mechanism to reuse the limited LB’s capacity. We always release the space consumed by an output tensor of a layer/block/sub-graph instance once it is no longer used. The space can thus be reused by other tensors. LB only needs to hold a limited number of tensors simultaneously. In case the total size of these tensors exceed LB’s capacity, the one with the longest liveness across multiple computation tasks is spilled to first GB and next DDR. Fig.12 depicts the liveness of tensors across computation tasks.

Our heuristic is different from prior work [40, 46], which always spills the tensor with the largest memory size to lower memory hierarchy levels. Selecting the one with the longest liveness has a higher probability to spill a smaller tensor, which is likely to reduce the overhead of data movements.

## 4.4 Across-layer Instruction Scheduling

Combining loop fusion and buffer stitching not only produces a single kernel for a sub-graph instance, but also allows for overlaps of different layer computation tasks. On the right of Fig.11, promoting a weight is implemented by first copying its tensor from DDR to GB using DMA and next hoisting the tensor from GB to LB, which is possibly further dispatched



(a) Execute  $ct_1$ . (b) Execute  $ct_4$ . (c) Execute  $ct_5$ . (d) Execute  $ct_7$ .

Figure 12: Liveness of tensors across computation tasks. A (ellipse) computation task ( $ct$ ) can be a sub-graph instance, a block or a layer. A (rectangle) tensor is live when colored in green or released if in gray. A  $ct$  is in execution if colored in yellow or finished when in gray. The space of tensor  $in_1$  is released once it is not live, reused by  $out_3$ .  $out_1$  is spilled to GB or DDR in case LB is insufficient to hold four tensors in (d), since it lives across seven  $cts$  but others across fewer.

to individual CUs of Fig.1. The latency of these promotion statements can be hidden behind an earlier executed layer, and multiple CUs can execute computation tasks simultaneously.

Fig. 13 shows how this optimization is performed. A rounded rectangular represents a layer’s tiled computation task. Across-layer memory latency hiding takes place between the two vertical lines, and CUs can execute different computation tasks of two tiles. Our approach significantly enhances the opportunities of such memory latency hiding and parallelism by increasing the optimization granularity to a degree beyond layers studied by prior work [5, 13, 31, 54].

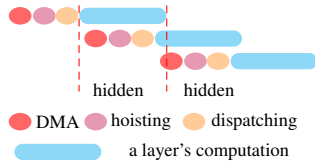


Figure 13: Across-layer instruction scheduling.

## 5 Experimental Results

We conduct experiments on STCP920 [51], an SoC DSA platform customizing the abstraction in Fig.1 using

$$\begin{cases} d \leftarrow 4; c \leftarrow 8; u \leftarrow 3 \\ LB \leftarrow 64 \text{ KB L1} \\ GB \leftarrow 8\text{MB last local buffer (LLB)} \\ CU_1 \leftarrow \text{vector core}; CU_2 \leftarrow \text{VME}; CU_3 \leftarrow \text{MME} \end{cases} \quad (2)$$

The eight cores connected bidirectionally using LLB. Each core has a 32-bit RISC-V CPU with vector extension, a vector MAC engine (VME) and a matrix MAC engine (MME) to handle different types of  $ops$ . As the target shares the common hardware abstraction of many existing DSA accelerators, one can expect similar results on other DSA platforms.

GraphTurbo resorts to LLVM v12.0.0 to compile its generated kernels on STCP920. The repository will be made publicly accessible soon. We experiment using ResNet-50 v1.5 [16], BERT [9], and DLRM [33], extracted from MLPerf v2.0 [43] and use their standard configurations. For BERT, we also consider three additional configurations. Other MLPerf models are not considered because they involve dynamically shaped tensors that GraphTurbo currently does not support. We also take into account MobileNet v2 [17], Vision\_Transformer [11], DenseNet [18], and Conformer [14] that are not included in MLPerf. Except DLRM implemented using Pytorch v1.8.1 [38], all remaining models are implemented using TensorFlow v1.13 [1]. There are no fundamental reasons that limit the applicability of GraphTurbo to training models, which we intend to experiment in the future.

GraphTurbo is prototyped in TVM v0.8 [5], implemented using 19k Python, 44.2k C/C++ and 2k miscellaneous, among which the code used to implement the graph scheduling approach is about 7k lines. As our algorithms operate on computational graphs, it does not require much effort to target GraphTurbo to a new platform. What the engineers need to do is to feed these algorithms with necessary architectural information required, and a target platform should share the same properties as the DSA abstraction in Fig.1. The code to be changed should be lightweight in such cases.

### 5.1 Task Decomposition across Clusters

We first discuss how the optimal batch size is selected for a cluster of STCP920 using BERT-128, whose sequence length is 128. Table 1 collects the data for both throughput and latency. We report the results of TVM v0.8 for this model, and also consider the result of highly-crafted C++ implementations provided by the vendor of STCP920, which schedules a computational graph in a similar way to our idea and implements kernel generation by hand.

Table 1: Results of BERT-128 under different batch sizes.

approach	batch size	configuration		throughput (sentences/s)	latency (ms)
		iter.	batches/cluster		
TVM	8	1	2	138	6.79
	16	1	4	512	9.48
	32	2	4	512	18.96
	64	4	4	512	37.92
GraphTurbo	8	1	2	138	6.79
	16	1	4	512	9.48
	32	1	8	4052	7.67
	64	1	16	2716	23.58
crafted code	64	2	8	4052	15.34
	32	1	8	4048	7.62

For TVM, a cluster’s LLB is sufficient to retain four data batches. As allocating four batches to each cluster makes two clusters idle, we allocate two batches to each cluster when the batch size is eight, which obtains a 138 sentences/s throughput and a 6.79ms latency. When the batch size increases to 16, TVM can allocate four batches to each cluster; the throughput and latency grow to 512 sentences/s and 9.48ms, respectively. As TVM cannot allocate more batches to a cluster, the throughput cannot further scale with the growth of the batch size. Instead, TVM introduces a loop execution within each cluster, which guarantees the throughput but the latency increases as proportional to the number of loop iterations.

GraphTurbo performs the same as TVM when the batch size is 8 and 16, since a cluster’s LLB is sufficient to handle the allocated batches and we do not need to create larger sub-graphs or split them. This illustrates that **the scheduling of TVM can be considered as a special case of our work**. The difference is observed when the batch size increases to 32, for which GraphTurbo allocates eight batches to a cluster but TVM only allocates four. GraphTurbo creates larger sub-graphs and splits them into instances, achieving a higher throughput of 4052 sentences/s and a lower latency of 7.67ms.

When the batch size increases to 64, GraphTurbo allocates 16 batches to each cluster but suffers from both throughput degradation and latency increase, since such a batch allocation requires larger tensors than the implementation in §4.3 spills more of them to slower buffers. In this case, we also introduce a loop execution within a cluster by allocating eight batches to it. As a result, the throughput stays at 4052 sentences/s and the growth of latency is also alleviated when compared with the case of allocating 16 batches to each cluster.

Table 1 also indicates that GraphTurbo achieves very close throughput and latency to the vendor-crafted implementation. In the following context, we report the results of TVM and GraphTurbo by selecting their optimal numbers of allocated batches for a cluster. Both optimal batch allocation strategies are obtained after a round of beforehand autotuning executions. For the sake of simplicity, we discuss throughout numbers below but the results also apply to latency.

## 5.2 Performance Comparison

We now report the performance. BERT is configured using four sequence lengths, 256, 384 (the default MLPerf configuration), and 512. Table 2 summarizes the configurations of each model. TVM’s throughput is listed in the rightmost column, which is preceded by the throughput units. We show the speedups of each approach over TVM’s data in Fig.14, where we also report the results of AStitch [60].

TVM still fuses *ops* within a sub-graph and produces kernels that exchange data via DDR, and it also misses the instruction scheduling opportunities across layers. By (1) producing fewer kernels and reducing off-core data movements, (2) better saturating L1, and (3) further exploiting across-

Table 2: Summary of the models.

label	model	batch size	batches per cluster		throughput unit	TVM’s result
			TVM	GraphTurbo		
(A)	ResNet-50	64	2	16	images/s	1064
(B)	BERT-128	32	4	8	sentences/s	512
(C)	BERT-256	16	2	4	sentences/s	412
(D)	BERT-384	8	1	2	sentences/s	36
(E)	BERT-512	8	1	2	sentences/s	324
(F)	DLRM	1024	64	256	queries/s	131000
(G)	MobileNet-v2	128	2	32	images/s	1416
(H)	Vision_Transformer	32	4	8	images/s	40
(I)	DenseNet	32	4	8	images/s	456
(J)	Conformer	12	1	3	sentences/s	184

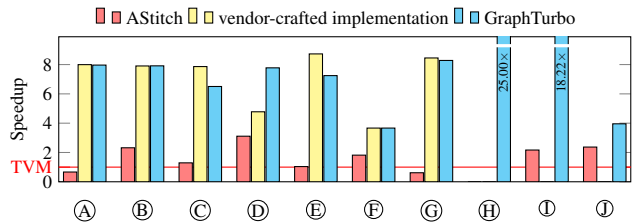


Figure 14: Speedups of throughput over TVM.

layer instruction scheduling, GraphTurbo outperforms TVM by 11.15× on average.

We reproduce the functionality of AStitch for STCP920 by maximally preserving tensors in L1. However, AStitch does not split a sub-graph into instances and thus fails to benefit from the imbalanced memory usage distribution enabled by better schedule orders of sub-graph instances. Hence, GraphTurbo obtains a mean speedup of 6.16× over it. Its performance falls behind that of TVM for ResNet-50 and MobileNet-v2 because AStitch prefers to produce a single kernel for compute-intensive *conv ops* in these models, which spills data to DDR and results in heavier data movements.

The vendor-crafted implementation considers all the optimization opportunities studied in this paper. Manually optimizing a computational graph can better exploit the trade-off between parallelism, load balance and locality, making crafted code sometimes obtain better performance than GraphTurbo, *e.g.*, for BERT-512 and MobileNet-v2. However, a manual scheduler is also non-trivial and thus sometimes misses the imbalanced memory usage distribution, *e.g.*, for BERT-384. Due to the complexity of such a scheduling strategy, the vendor implementation for the last three models is still under construction till now, and their data are thus missing. On average, GraphTurbo achieves a 1.04× speedup over the vendor-crafted implementation.

## 5.3 Performance Breakdown

This section studies how different factors contribute to the overall speedup of GraphTurbo over TVM. We consider four variants of GraphTurbo as follows. First, we only keep the outputs of each kernel generated by GraphTurbo in LLB as



much as possible. Second, we let these outputs stay in L1 to the greatest extent. Third, we split sub-graphs into instances and schedule them based on the second variant, but across-layer instruction scheduling is disabled. Finally, we turn on all optimizations. Fig.15 shows the comparison results.

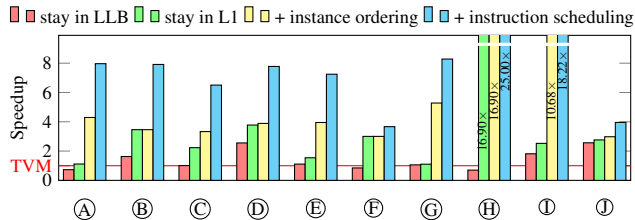


Figure 15: Individual contributions of each optimization.

The results of the second variant illustrate that converting off-core data movements into on-core data exchanges indeed makes sense. By achieving an average speedup of  $3.67\times$  over TVM, the green bars also outperform the red bars, which do not always obtain positive speedups over TVM. This demonstrates the importance of preserving tensors greedily in LB, *i.e.*, L1 of STCP920.

The third variant outperforms the green bars by  $2.20\times$  on average. ResNet-50, BERT-256, BERT-512, MobileNet-v2, and DesneNet, that exhibit imbalanced memory usage distribution caused by their down-sampling operations, benefit more from the ordering of sub-graph instances. Other models that either do not have such a network property (BERT-128 and Vision\_Transformer) or introduce more concatenation *ops* than the remaining ones (DLRM) observe insignificant improvements. Finally, across-layer instruction scheduling (§4) obtains a mean speedup of  $1.72\times$  over the third variant.

## 5.4 Hardware Utilization

This section evaluates how effectively GraphTurbo can utilize DSA hardware resources. First, as the core idea is to convert off-core data movements to on-core data exchanges, we investigate how the memory hierarchy of STCP920 is utilized. To this end, we report in Table 3 the frequencies of each memory level that different approaches utilize.

Table 3: Comparison of buffer scopes.

label	DDR			LLB			L1		
	TVM	crafted	GraphTurbo	TVM	crafted	GraphTurbo	TVM	crafted	GraphTurbo
(A)	58	1	1	0	11	11	0	291	284
(B)	242	2	1	0	0	0	0	304	305
(C)	242	2	1	0	25	110	0	401	240
(D)	515	2	1	0	49	75	0	968	967
(E)	242	2	1	0	25	76	0	474	337
(F)	76	1	0	0	0	0	0	75	76
(G)	56	1	0	0	7	3	0	619	608
(H)	214	-	24	0	-	60	0	-	340
(I)	247	-	0	0	-	3	0	-	389
(J)	1054	-	4	0	-	813	0	-	250

TVM always puts the output tensors of its sub-graphs in DDR, resulting in abundant off-core data movements. In contrast, GraphTurbo maximizes the utilization of faster local memory, converting many off-core data movements into on-core data exchanges. The vendor-crafted implementation also makes use of the faster local memory. Due to their familiarity with the hardware, the architects of STCP920 sometimes can better manage the memory hierarchy than our heuristics, but this manual scheduler is also tedious.

Second, we evaluate how VME and MME are utilized using ResNet-50 and BERT-128. We report in Fig.16 the data under different batch sizes, with the quantization version of ResNet-50 also considered, to validate the scalability of GraphTurbo. Other models observe similar results. The utilization of both VME and MME increases with the growth of batch size, which is exploited by our work. BERT-128 suffers from a degradation when the batch size changes from 8 to 16, as explained in §5.1.

Fig.17 shows the utilization of VME and MME when executing the four stages in Fig.3a. GraphTurbo performs similar to TVM for stage1, but it outperforms TVM for the other three stages, which demonstrates exploiting the imbalanced memory usage distribution can better utilize hardware resources.

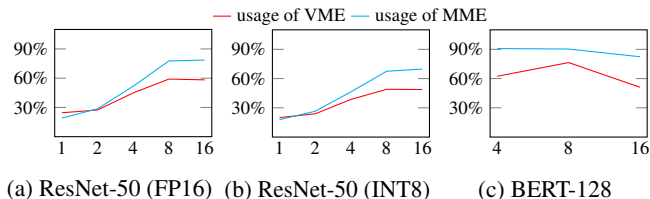


Figure 16: Usage of VME/MME. x axis denotes batch sizes.

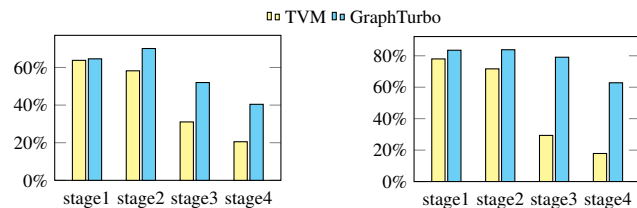


Figure 17: Utilization VME (left) and MME (right) when executing the stages in Fig.3a. y axis is the utilization percentage.

## 5.5 Comparison of Compilation Overhead

As compilation time is also a major concern for scheduling DNN models, this section reports the compilation overhead of GraphTurbo and compares it with those of the baseline methods. Table 4 reports the data in seconds, which demonstrates that GraphTurbo can achieve better performance than the state of the art without significantly aggravating the compilation overhead.

Table 4: Comparison of compilation overhead in seconds.

label	TVM	AStitch	GraphTurbo
(A)	102	66	139
(B)	159	128	199
(C)	170	136	224
(D)	312	290	699
(E)	171	143	282
(F)	25	22	23
(G)	74	57	248
(H)	189	146	340
(I)	173	129	189
(J)	382	238	296

## 5.6 Case Study on GPU

We now conduct a case study using the ResNet18-Tailor model to validate that scheduling sub-graph instances can be extended to NVIDIA A100 GPU. We use CUTLASS v2.9 [27] to implement kernel generation, which is compiled using CUDA toolkit v11.4 with `-O3` flag. We did not consider the *conv* and maxpooling *ops* at the start, and avgpooling and softmax *ops* at the end of this model since they do not contribute to imbalanced memory usage distribution, like those before and after the stages of Fig. 2. The other layers, each composed of a *conv* and ReLU *ops* using a circle, are shown in Fig. 18a.

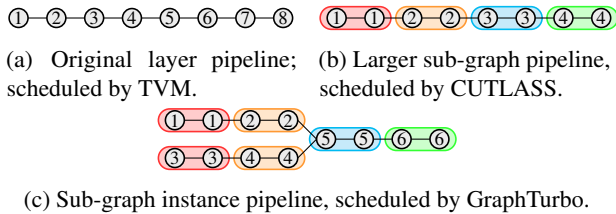


Figure 18: Different schedule orders of ResNet18-Tailor. The numbers define the schedule order of each method.

We let TVM wrap CUTLASS when generating kernels for layers. CUTLASS fuses two *conv* layers, partitioning the model into four larger sub-graphs (Fig. 18b), the first two of which is split by GraphTurbo into two instances (Fig. 18c). The output tensors of the (red) sub-graph instances are stitched via GPU registers and those of the (orange) ones through GPU shared memory. Table 5 summarizes the results.

Table 5: Execution time in milliseconds on A100 GPU.

batch size	TVM	CUTLASS fusion	graph scheduling	Speedup over	
				TVM	CUTLASS fusion
64	0.99	0.84	0.83	1.19×	1.01×
128	1.88	1.62	1.50	1.25×	1.08×
256	3.51	3.04	2.83	1.24×	1.07×
512	6.76	5.83	5.44	1.24×	1.07×
average				<b>1.23×</b>	<b>1.06×</b>

By scheduling sub-graph instances and exploiting GPU registers and shared memory between them, GraphTurbo outperforms TVM by 1.23× on average, which demonstrates that

our idea is also useful on GPU. GraphTurbo also achieves a mean speedup of 1.06× over CUTLASS, because scheduling sub-graph instances also brings about benefits by exposing and exploiting imbalanced memory usage distribution.

The reasons why the performance improvements on GPU is not promising as on STCP920 are two-folded. On the software side, GraphTurbo can construct a larger sub-graph that contains more than two layers, but CUTLASS, which we use for kernel generation here, refused to accept three or more *conv* layers. Enhancing the kernel generator of TVM in the future can address this issue. We also did not apply instruction scheduling here. On the hardware side, the higher memory bandwidth of this GPU also makes the improvements caused by reducing off-core data movements not significant as on STCP920. While STCP920 only delivers a memory bandwidth of 136GB/s, this GPU can reach more than 1500GB/s.

Nonetheless, other NVIDIA software-defined platforms with limited memory bandwidth, e.g., NVIDIA DRIVE AGX Orin [36], could benefit from GraphTurbo. We also believe that the software-controlled inter-cooperative-thread-array shared memory of the latest NVIDIA H100 GPU could be better exploited by the idea presented in this paper. Hence, our work also offers insights to the GPU micro-architectures.

## 6 Related Work

Scheduling its computational graph is the first step to deploy a DNN model on platforms. The difference between our scheduler and prior work is that we consider hardware architecture when grouping sub-graphs, which enables the synergy between network architecture and DSA, while existing methods [13, 19, 44, 49] not. By generating coarser-grained sub-graphs and splitting them into instances, GraphTurbo exposes the imbalanced memory usage distribution, a network property first studied by MCUNetV2 [30]. However, MCUNetV2 only discusses tiny DNN models on microcontroller units, while this paper considers large-scale DNN models and targets a cloud DSA chip. Some optimizations that can only be implemented when sub-graphs are lowered to loop nest pipelines are not considered by MCUNetV2 but studied in §4.

When a sub-graph is lowered to a loop nest pipeline, existing methods like TVM [5] fuse loop nests with the help of manually written schedule templates. As TVM does not scale well with the increase of *op* numbers within a sub-graph, we only use TVM’s loop fusion to group loop nests within a layer. Our implementation also avoids the need to manually write schedule templates and inject memory promotion statements of weight tensors, the later of which is automated by interacting with the graph scheduler.

By expanding a high-level sub-graph into individual low-level *ops*, XLA [13] does not restrict fusion within layers. Nonetheless, retrieving the high-level information via low-level *ops* is critical to fuse low-level *ops* for XLA, and manually forming profitable high-level sub-graphs is considered as

more robust than through automatic pattern matching in this compiler [45]. GraphTurbo fully automates this process and achieves better performance than AStitch, which has already been demonstrated as superior to XLA [60].

IREE [20] is another work that makes use of its graph scheduling logic when communicating data between low-level parallel pipelined hardware/APIs. Our work differs from IREE by focusing on scheduling instances of larger sub-graphs, which tends to produce fewer kernels. By managing an internal map of *op* sequence on the fly, Zero-Infinity [42] exploits the fine-grained overlaps by prefetching the parameters required by future *ops* during the execution of the current *op*. A layer considered by our work usually includes multiple *ops*, the execution of which is more likely to fully hide the data transfer overhead.

Some autotuning frameworks [6, 57, 59] are also devised to enhance the power of TVM with fewer or no hand-written schedule templates. These autotuners use their search heuristics to tune memory optimizations to further improve the performance of their generated code. Unfortunately, their search spaces are all restricted within layers [39], while our work enables across-layer instruction scheduling (§4.4). In particular, Ansor [57] represents the state of the art of this kind, which is orthogonal to our work by neither considering the scheduling of GraphTurbo nor exploiting the fusion possibilities of multiple *conv*/matrix multiplication operators. Loop fusion is also investigated by polyhedral frameworks [3, 47, 54], but they did not consider buffer stitching that has been discussed in §4.2. Similar to GraphTurbo, DNNFusion [34] also studies across-layer fusion for mobile devices. We fail to obtain its repository to conduct an experimental comparison.

Another thread of works [10, 28, 35] investigate horizontal fusion between *ops* with no producer-consumer relations to better utilize the hardware resources of their targets. GraphTurbo tackles the same issue using a different idea. Our scheduler exploits parallelism within a sub-graph instance, which is always homogeneous to other instances of the same sub-graph. It always decomposes one or multiple dimensions of a tensor to exploit parallelism. On the contrary, *ops* grouped by horizontal fusion are heterogeneous, which calls for a more complicated parallelization mechanism.

Recently, schedulers and code generators for DSA platforms are widely studied. Rammer [31] and Roller [62] generate code for Graphcore IPU [22]. They maximize the utilization of faster memory by combining *ops* that cannot saturate hardware resources. AKG [54] targets code generation for Ascend 910 [29] using the polyhedral model [4, 48] to perform loop fusion. XLA [13] and NaaS [61] exploit the scheduling of sub-graphs for generating code on TPU [25].

The distinction between our work and these approaches is that GraphTurbo partitions a sub-graph along one output tensor's dimension while these methods partition tensors by tiles along multiple dimensions. The primary reason why GraphTurbo does this way is because cores in the DSA ab-

straction of Fig.1 are organized in 1D form. For instance, this level corresponds to 32 hardware cores sharing the shared memory of GPU. The partitioning approach of GraphTurbo is also extensible to deal with a multi-dimensional core grid organization by gradually partitioning and mapping multiple loop dimensions to these hardware dimensions. Moreover, as their targets share the DSA abstraction in Fig.1, the idea presented in this paper could also be used on their targets.

## 7 Conclusion

GraphTurbo is a scheduler for DNN models that enables the synergy between network and hardware architectures. This significant difference from prior work produces fewer kernels and thus reduces off-core data movements, better saturates faster local memory of DSA platforms by exploiting the imbalanced memory usage distribution, and opens opportunities for across-layer instruction scheduling. Results of seven DNN models demonstrate the effectiveness of our idea, whose applicability to GPU is also discussed.

GraphTurbo obtains sub-graph instances by selecting an appropriate size to split a DNN computation graph. Indeed, selecting the optimal size to perfectly model a DSA's memory hierarchy is challenging, and only making use of LB in Fig.1 is not the optimal solution. Instead, our method is just a greedy idea that has been demonstrated effective when compared with vendor-crafted implementation, which we believe can be considered as a good result. A more intelligent approach can be explored to catch up or even beat the performance obtained by hand for models like BERT-512 and MobileNet-v2.

GraphTurbo currently has two limitations. First, the optimal batch size for a cluster is still selected by a simple autotuning approach. We intend to develop an intellectual technique to better address this issue. Second, GraphTurbo cannot handle dynamically shaped tensors. Integrating with the recent methods [12, 56] along this direction may alleviate this issue.

## Acknowledgments

We acknowledge Hyeontaek Lim for his shepherding and the OSDI'23 reviewers for their constructive comments that improve the quality of this work. We would also like to express our gratitude to Bojie Li and Jun Yang for their suggestions on the early versions of this paper, Zhongzhou Jiang, Yuqing Wang, Di Mei, and many other toolchain team members of the Streaming Computing Inc. for their help during the use of their vendor-crafted implementation. Jie Zhao and Xiaoqiang Dan are the corresponding authors of this paper, and the work of Jie Zhao is partly supported by the National Natural Science Foundation of China under Grant No. U20A20226. The views and conclusions presented in this paper belong to the authors. Interpreting them as the official policies of the Chinese Government in any way is not acceptable.

## References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.
- [2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.
- [3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019*, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.
- [4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, pages 101–113, New York, NY, USA, 2008. ACM.
- [5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.
- [6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.
- [7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.
- [8] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, oct 2016.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- [10] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 1–14, 2021.
- [11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.
- [12] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 721–747, 2022.
- [13] Google. Xla: Optimizing compiler for machine learning. <https://www.tensorflow.org/xla>, 2017.
- [14] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented Transformer for Speech Recognition. In *Proc. Interspeech 2020*, pages 5036–5040, 2020.
- [15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.
- [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.



- [17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.
- [19] Intel. onednn graph specification 1.0-alpha. <https://spec.oneapi.io/onednn-graph/latest/index.html>, 2020.
- [20] IREE. Iree. <https://iree-org.github.io/iree/>, 2021.
- [21] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefer. Data movement is all you need: A case study on optimizing transformers. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 711–732, 2021.
- [22] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.
- [23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP’19*, pages 47–62, New York, NY, USA, 2019. ACM.
- [24] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, aug 2018.
- [25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gotipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA’17*, pages 1–12, New York, NY, USA, 2017. ACM.
- [26] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 387–400, 2021.
- [27] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++. <https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/>, 2017.
- [28] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8343–8354. Curran Associates, Inc., 2020.
- [29] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801, 2021.
- [30] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Memory-efficient patch-based inference for tiny deep learning. In M. Ranzato, A. Beygelzimer, K. Nguyen, P.S. Liang, J.W. Vaughan, and Y. Dauphin, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 1–13. Curran Associates, Inc., 2021.
- [31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.
- [32] Eitan Medina. Habana labs presentation. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–29, 2019.

- [33] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.
- [34] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, pages 883–898, New York, NY, USA, 2021. ACM.
- [35] NVIDIA. Nvidia tensorrt. <https://developer.nvidia.com/tensorrt>, 2016.
- [36] NVIDIA. Nvidia introduces drive agx orin — advanced, software-defined platform for autonomous machines. <https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orin-advanced-software-defined-platform-for-autonomous-machines>, 2017.
- [37] Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 584–597, 2021.
- [38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.
- [39] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Reza Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Rouné, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16, 2021.
- [40] Yury Pisarchyk and Juhyun Lee. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288*, 2020.
- [41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI’13*, pages 519–530, New York, NY, USA, 2013. ACM.
- [42] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’21*, New York, NY, USA, 2021. Association for Computing Machinery.
- [43] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Isgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Genady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA ’20*, pages 446–459. IEEE Press, 2020.
- [44] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.
- [45] Bjarke Rouné. Compiling ml with xla (slides). <https://www.c4ml.org/c4ml2019>, pages 16–24, 2019.
- [46] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

- [47] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Trans. Archit. Code Optim.*, 16(4), October 2019.
- [48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.
- [49] Richard Wei, Lane Schwartz, and Vikram Adve. DlvM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.
- [50] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.
- [51] Rongkai Zhan and Xiaobo Fan. Neuralscale: A risc-v based neural processor boosting ai inference in clouds. In *Fifth Workshop on Computer Architecture Research with RISC-V, CARRV*, 2021.
- [52] Jie Zhao and Peng Di. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture, MICRO-53*, pages 427–441, Piscataway, NJ, USA, 2020. IEEE Press.
- [53] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 1–19, 2022.
- [54] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI’21*, pages 1233–1248, New York, NY, USA, 2021. ACM.
- [55] Yongwei Zhao, Zidong Du, Qi Guo, Shaoli Liu, Ling Li, Zhiwei Xu, Tianshi Chen, and Yunji Chen. Cambricon-f: Machine learning computers with fractal von neumann architecture. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA’19*, pages 788–801, New York, NY, USA, 2019. ACM.
- [56] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 848–863, 2022.
- [57] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.
- [58] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA ’22*, pages 874–887, New York, NY, USA, 2022. Association for Computing Machinery.
- [59] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS’20*, pages 859–873, New York, NY, USA, 2020. ACM.
- [60] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022*, pages 359–373, New York, NY, USA, 2022. ACM.
- [61] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Towards the co-design of neural networks and accelerators. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 141–152, 2022.

[62] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER:

Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.





# EINNET: Optimizing Tensor Programs with Derivation-Based Transformations

Liyang Zheng<sup>◇</sup> Haojie Wang Jidong Zhai Muyan Hu Zixuan Ma Tuowei Wang  
 Shuhong Huang Xupeng Miao<sup>†</sup> Shizhi Tang Kezhao Huang Zhihao Jia<sup>†</sup>  
*Tsinghua University* <sup>†</sup>*Carnegie Mellon University*

## Abstract

Boosting the execution performance of deep neural networks (DNNs) is critical due to their wide adoption in real-world applications. However, existing approaches to optimizing the tensor computation of DNNs only consider transformations *representable* by a fixed set of predefined tensor operators, resulting in a highly restricted optimization space. To address this issue, we propose EINNET, a *derivation-based* tensor program optimizer. EINNET optimizes tensor programs by leveraging transformations between *general* tensor algebra expressions and automatically creating new operators desired by transformations, enabling a significantly larger search space that includes those supported by prior works as special cases. Evaluation on seven DNNs shows that EINNET outperforms existing tensor program optimizers by up to  $2.72\times$  ( $1.52\times$  on average) on NVIDIA A100 and up to  $2.68\times$  ( $1.55\times$  on average) on NVIDIA V100. EINNET is publicly available at <https://github.com/InfiniTensor/InfiniTensor>.

## 1 Introduction

Fast execution of deep neural networks (DNNs) is critical in a variety of tasks, such as autonomous driving [16, 21, 26], object detection [15, 18], speech recognition [5, 17], and machine translation [37, 39]. A DNN is generally represented as a *tensor program*, which is a directed acyclic graph containing tensor operators (e.g., convolution, matrix multiplication) performed on a set of tensors (i.e.,  $n$ -dimensional arrays).

To improve the runtime performance of a DNN, existing frameworks (TensorFlow [3], PyTorch [31], and TensorRT [35]) rely on *manually-designed* rules to map an input tensor program to *expert-written* kernel libraries. Although widely used, these approaches require extensive engineering efforts and miss optimization opportunities hard to manually discover. To address these problems, recent works have proposed a variety of *automated* approaches that optimize DNN computation by searching over a set of candidate

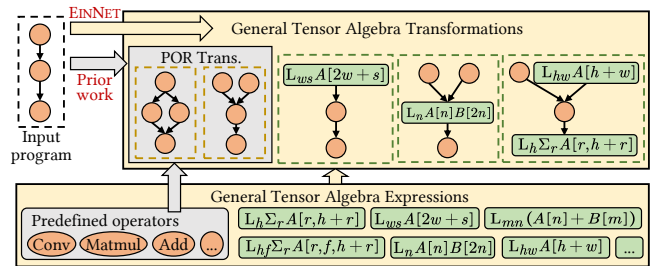


Figure 1: Comparing EINNET’s search space with that of prior work. “POR Trans.” indicates predefined operator representable transformations.

program transformations or generating high performance kernels on specific hardware. We classify these works into two categories based on their search spaces.

The first category of work, including TVM [7] and Anso [40], is motivated by Halide’s idea of compute/schedule separation [33] and optimizes tensor programs at the *operator level*. For a given tensor operator, they automatically generate high-performance kernels by searching over *schedules*, each of which specifies an architecture-dependent execution plan on particular hardware. To optimize the graph structure of a tensor program, TVM and Anso greedily apply a fixed set of expert-designed program transformations.

The second category of work optimizes tensor programs using *graph-level* transformations, which reorganize the DNN computation in more efficient ways. As two representative systems, TASO [20] and PET [38] adopt a superoptimization-based approach to discovering graph transformations. They generate candidate graph transformations by enumerating all possible graphs over a given set of tensor operators up to a fixed size, and search to apply these generated transformations to an input tensor program.

Both operator- and graph-level optimizers only consider program transformations whose nodes are tensor operators predefined by optimizer developers, as shown in the grey box of Figure 1. We call these transformations *predefined operator representable* (POR) transformations. Despite the fact that

<sup>◇</sup>Tsinghua University and BNRist

existing tensor program optimizers only use POR transformations to optimize tensor programs, POR transformations only exhibit limited opportunities for performance optimizations. In this paper, we propose to explore *general* tensor algebra transformations whose nodes are *general* tensor operators<sup>1</sup>. Compared to POR transformations, general tensor algebra transformations constitute a significantly larger optimization space, which includes POR transformations as special cases, as shown in the yellow box of Figure 1.

To discover general tensor algebra transformations, we present EINNET, a *derivation-based* tensor program optimizer. A key difference between EINNET and prior work (e.g., TASO and PET) is that EINNET reveals operator computation semantics in automated graph transformations by applying derivation rules to tensor algebra expressions. By deriving computation at the expression level, EINNET can reorganize computation into arbitrary tensor expressions and map them into both predefined operators with highly optimized implementations and new auto-generated operators desired by derivations. Expression-level derivations allow EINNET to discover a variety of novel program transformations missing in existing frameworks, since these transformations require highly customized tensor operators not predefined in existing optimizers. Example transformations newly discovered by EINNET include: (1) modifying the computation semantics of an operator to improve efficiency, (2) replacing inefficient operators with highly-optimized alternatives and customized tensor operators to bridge the gap, and (3) aggressively reorganizing computation graphs to enable subsequent graph-level optimizations.

EINNET mainly addresses the following three challenges:

The first challenge is automatically discovering transformation opportunities between general expressions. TASO and PET only consider a *fixed* set of predefined operators, but there are infinitely many possible general expressions. Hence, directly applying superoptimization (i.e., enumerating all possible graphs over general expressions) is infeasible. EINNET addresses this challenge by presenting a *derivation-based* mechanism that automatically transforms an expression to equivalent alternatives by applying a collection of derivation rules. Since most derived expressions cannot be simply represented as predefined operators, we introduce *eOperators* (*expression as an operator*) to represent non-POR computation. eOperators enable EINNET to discover a variety of optimizing transformations between expressions.

The second challenge is converting expressions back to kernels that can be executed on DNN accelerators, a process we term *expression instantiation*. Although existing kernel generators (e.g., TVM and Ansor) can generate kernels for a given expression, doing so is suboptimal since existing vendor-provided libraries (e.g., cuDNN [10] and cuBLAS [11]) offer highly-optimized kernels for a set of

<sup>1</sup>An operator is a tensor operator if it can be represented using the tensor algebra expression in Equation (1)

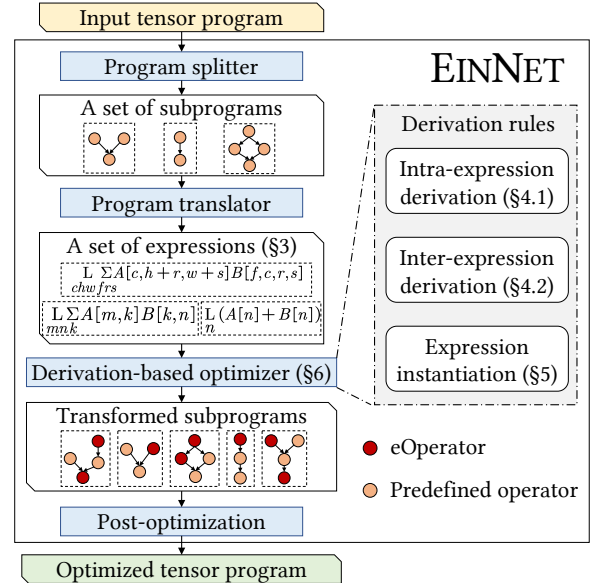


Figure 2: EINNET overview

predefined operators. EINNET opportunistically matches a part of an expression with predefined operators to take advantage of the highly-optimized kernels from vendor-provided libraries; the remaining part of the expression is lowered to an off-the-shelf kernel generator (i.e., TVM [7]).

The third challenge is quickly finding optimizing transformations in the search space of general tensor algebra transformations. In particular, optimizing a tensor program normally requires applying a long sequence of derivation rules (e.g., up to 12 in our evaluation), which cannot be efficiently discovered by a traversal-based search algorithm. To address this challenge, EINNET employs a two-stage search approach to applying derivations, where an *explorative derivation* stage considers applying all possible derivations to the current expression to create a comprehensive collection of expressions, and a *converging derivation* stage uses *expression distance* to guide the search towards promising candidates. This distance-guided approach allows EINNET to discover complex optimizations requiring long sequences of derivations under a reasonable search budget.

We evaluate EINNET on seven real-world DNN models covering a variety of machine learning tasks. We compare EINNET with state-of-the-art frameworks on two GPU platforms, NVIDIA A100 and V100. Evaluation shows that EINNET is up to  $2.72\times$  faster than existing tensor program optimizers. The significant performance improvement indicates that EINNET benefits from the new optimization opportunities enabled by derivation-based optimizations.

This paper makes the following contributions:

- We extend the POR optimization space to the general tensor algebra optimization space by combining operator computation semantics and computation graphs with tensor algebra expressions.

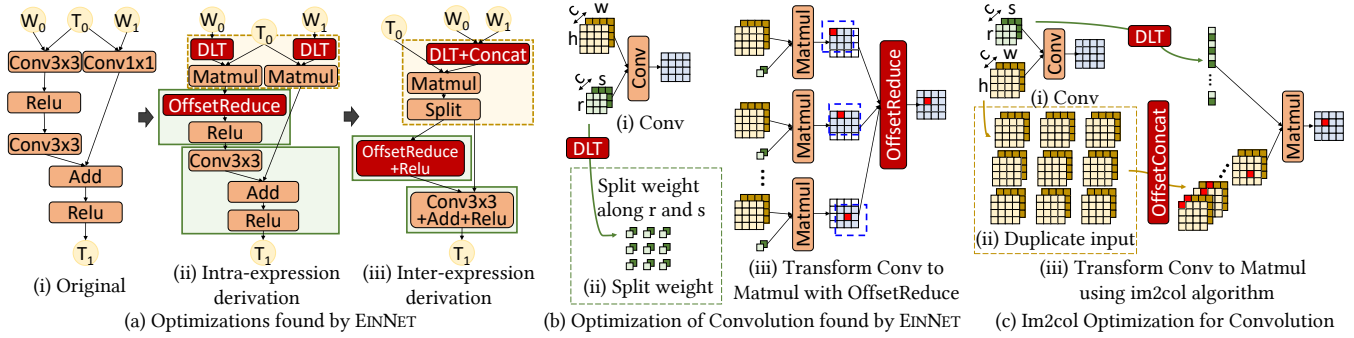


Figure 3: Optimization examples of EINNETH. Figure (a) shows the optimization that transforms a  $\text{Conv}3 \times 3$  operator into a  $\text{Matmul}$  and an eOperator  $\text{OffsetReduce}$ , and a  $\text{Conv}1 \times 1$  operator into a  $\text{Matmul}$ . Then, inter-expression derivation is performed to fuse multiple operators into one. Figure (b) shows the optimization details performed by EINNETH for the  $\text{Conv}3 \times 3$  operator, which first splits the weight tensor into 9 tensors, then multiplies each tensor with the input, and finally adds the nine results together with certain offsets (illustrated by the dashed boxes and red blocks). The  $\text{Matmul}$ s in Figure (b) are further fused into a single one. As a comparison, Figure (c) shows the typical  $\text{im}2\text{col}$  [36] optimization for  $\text{Conv}$ , which performs a different transformation from that in Figure (b) and can also be automatically found by EINNETH.

- We present the first attempt to explore a significantly larger expression search space using a derivation-based mechanism.
- We build EINNETH, an implementation of the above techniques with over 23K lines of C++ and Python code, which achieves up to  $2.72 \times$  speedup over existing tensor program optimizers.

## 2 Overview and Motivating Example

Figure 2 shows an overview of EINNETH, a tensor program optimizer with derivation-based transformations. For an input tensor program, EINNETH first splits it into multiple subprograms consisting of predefined operators. Each subprogram is translated to a tensor algebra expression (§3) by a program translator. Then, EINNETH’s *derivation-based optimizer* uses different derivation rules, including inter- and intra-expression derivation rules (§4) and expression instantiation rules (§5), to generate optimized subprograms for each expression, which consists of both predefined operators and eOperators. Finally, EINNETH selects the best discovered transformation for each subprogram and post-optimizes the expressions to construct an efficient tensor program (§6).

**Motivating example.** As a motivating example, Figure 3(a) shows an optimization found by EINNETH. It first performs an intra-expression derivation to transform convolutions into matrix multiplications, and then performs inter-expression derivation to fuse multiple operators into one. The red operators, such as  $\text{OffsetReduce}$ ,  $\text{DLT}$  (data layout transformation), and  $\text{OffsetReduce+Relu}$ , are eOperators automatically discovered and generated by EINNETH. Figure 3(b) shows the details of the new optimization discovered by EINNETH for

$\text{Conv}3 \times 3$  in Figure 3(a). Figure 3(c) illustrates the classic  $\text{im}2\text{col}$  [36] optimization for convolution, which is widely implemented in existing libraries and also covered by the automatic optimization space of EINNETH. Different from copying input tensors for the kernel size times in  $\text{im}2\text{col}$ , the newly discovered transformation copies output tensors the same number of times. It can be more efficient when the output size is smaller than the input size, and achieves a  $2 \times$  speedup compared with  $\text{cuDNN}$  on the NVIDIA A100 GPU for certain convolutions in ResNet-18 [19] in our evaluation.

Existing tensor program optimizers cannot automatically discover such transformations because: (1) the transformations require eOperators (e.g., adding intermediate tensors with offsets), which are outside of the POR transformation space explored by superoptimization-based frameworks such as TASO [20] and PET [38], and (2) the transformations modify the computation semantics instead of the schedule, and thus cannot be found by schedule-based optimizers like TVM [7] and Ansor [40].

## 3 Tensor Algebra Expression

EINNETH represents a tensor program as *tensor algebra expressions*, which defines how to compute each element of output tensors from input tensors. Figure 4 shows the expression of multiplying three matrices (i.e.,  $A \times B \times C$ ). We now describe the components of an expression. For simplicity, we assume an expression has one output. EINNETH’s expression can be easily generalized to multiple outputs.

**Traversal and summation notations.** A *traversal notation*, denoted as  $L_{x=x_0}^{x_1}$ , consists of an *iterator*  $x$  and an *iterating space*  $[x_0, x_1)$ . The traversal notation corresponds to a dimen-



$$\begin{aligned}
& \prod_{c=0}^C \prod_{r=0}^R \sum_{k_0=0}^K \sum_{k_1=0}^K A[c, k_0] B[k_0, k_1] C[k_1, r] \quad (a) \\
& = \prod_{c=0}^C \prod_{r=0}^R \sum_{k_0=0}^K \left\{ \prod_{c'=0}^C \prod_{k_2=0}^K \sum_{k_1=0}^K A[c', k_1] B[k_1, k_2] \right\} [c, k_0] C[k_0, r] \quad (b)
\end{aligned}$$

Traversal notation    Summation notation    Scope

Figure 4: A tensor algebra expression example for two matrix multiplications  $A \times B \times C$ . The red box highlights a *scope* that instantiates the intermediate result of  $A \times B$ .

sion of the output tensor, where the iterating space is the range of the dimension. The order of the traversal notations indicates the layout of the output tensor. For example, in Figure 4,  $\prod_{c=0}^C$  followed by  $\prod_{r=0}^R$  indicates that the expression’s output is a two-dimensional tensor with a shape  $C \times R$ .

A *summation notation*, denoted as  $\sum_{x=x_0}^{x_1}$ , computes the summation iterating over dimension  $x$  with  $\{x_0, x_0 + 1, \dots, x_1 - 1\}$ , which is hereinafter represented by a range  $[x_0, x_1]$  for brevity. Note that an EINNET expression under different orders of summation notations are considered the same but corresponds to different *schedules* of an expression. Therefore, it is excluded from the expression search space.

Tensors are indexed by an *arithmetic combination* of multiple iterators, including *add*(+), *sub*(-), *mul*(\*), *div*(/) and *mod*(%). For simplicity, we may merge multiple iterators into an iterator vector, whose iterating space can be denoted by an integer set or omitted in the expression. For example,  $\prod_{c=0}^C \prod_{r=0}^R$  can be represented as  $\prod_{cr}$  or  $\prod_{\vec{x}}$ , where  $\vec{x} = (c, r)$  is the iterator vector, and  $\mathbb{X} = \mathbb{C} \times \mathbb{R}$  is the iterating space.

**Scope.** For a tensor program with multiple operators (e.g., two consecutive matrix multiplications  $A \times B \times C$ ), a common optimization is to instantiate and reuse intermediate results (e.g., caching the output of  $A \times B$ ), which avoids repetitive computation for these results. EINNET introduces *scopes* to represent the instantiation of intermediate results to reuse them later. Formally, a tensor algebra expression is a *scope*, denoted by a surrounding  $\{\}$ , if the output of the expression is instantiated into a tensor, which allows subsequent computation to refer to this tensor and therefore avoids repeated computation. In Figure 4(b), the expression corresponding to  $A \times B$  is a scope, allowing subsequent computation to directly refer to the output of this expression. Many of EINNET’s derivation rules are based on transformations between scopes, including generating new scopes from existing ones, transforming a scope to another form, and merging multiple scopes into one (§4). Transformations between scopes are essential to EINNET’s optimizations.

**Padding.** Some computations access an input outside of its region, which we call paddings. E.g., a  $3 \times 3$  convolution may have paddings. Paddings are set to 0 if not specified.

**General format.** We represent a one-scope expression as:

Table 1: Derivation rules for tensor algebra expressions.

Rules	Descriptions
Intra-expression derivation §4.1	
Summation splitting	Split summation from one scope into two
Variable substitution	Replace traversal iterators with new ones
Traversal merging	Merge two scopes by merging traversals
Boundary relaxing	Relax the range of iterators
Boundary tightening	Tighten the range of iterators
Inter-expression derivation §4.2	
Expression splitting	Split an expression into independent ones
Expression merging	Merge multiple independent expressions
Expression fusion	Fuse multiple dependent expressions
Expression instantiation §5	
Operator matching	Match a scope with predefined operators
eOperator generation	Generate an eOperator for a scope

$$\prod_{\vec{x}} \sum_{\vec{y}} f(\mathbf{T}[\tau(\vec{x}, \vec{y})]) \quad (1)$$

where  $\mathbf{T} = \{T_0, T_1, \dots\}$  is a list of input tensors,  $\tau(\vec{x}, \vec{y})$  is the indexing function that computes element indexes for tensors in  $\mathbf{T}$  using iterators  $\vec{x}$  and  $\vec{y}$ , and  $f$  is the computation taking on the indexed elements of  $\mathbf{T}$ .

## 4 Derivation Rules

To discover highly-optimized expressions for an input tensor program, EINNET uses *derivation rules* to apply transformations on an input expression. Table 1 summarizes the derivation rules used by EINNET. Note that the *mathematical equivalence* of derivation rules guarantees the equivalence of derived expressions discovered by EINNET.

Different from schedule primitives of kernel generators that are designed to discover optimized schedules of a given expression on specific hardware, EINNET’s derivation rules focus on transform the computation semantics of tensor expressions, such as reorganizing computation into efficient operators.

### 4.1 Intra-Expression Derivation

Intra-expression derivation rules transform an expression into other functionally equivalent forms, which is essential for constructing a comprehensive search space of expressions for a tensor program. Figure 5 shows the optimization details in Figure 3(b). It splits the expression of `Conv3x3` into two parts, derives one part toward a predefined operator `Matmul`, and then converts the other part to an eOperator. We now describe these intra-expression derivation rules.

**Summation splitting** divides a summation notation  $\sum_{\vec{s}}$  into two separate summations  $\sum_{\vec{s}_1}$  and  $\sum_{\vec{s}_2}$  and instantiates the result of the inner summation by converting it to a scope:

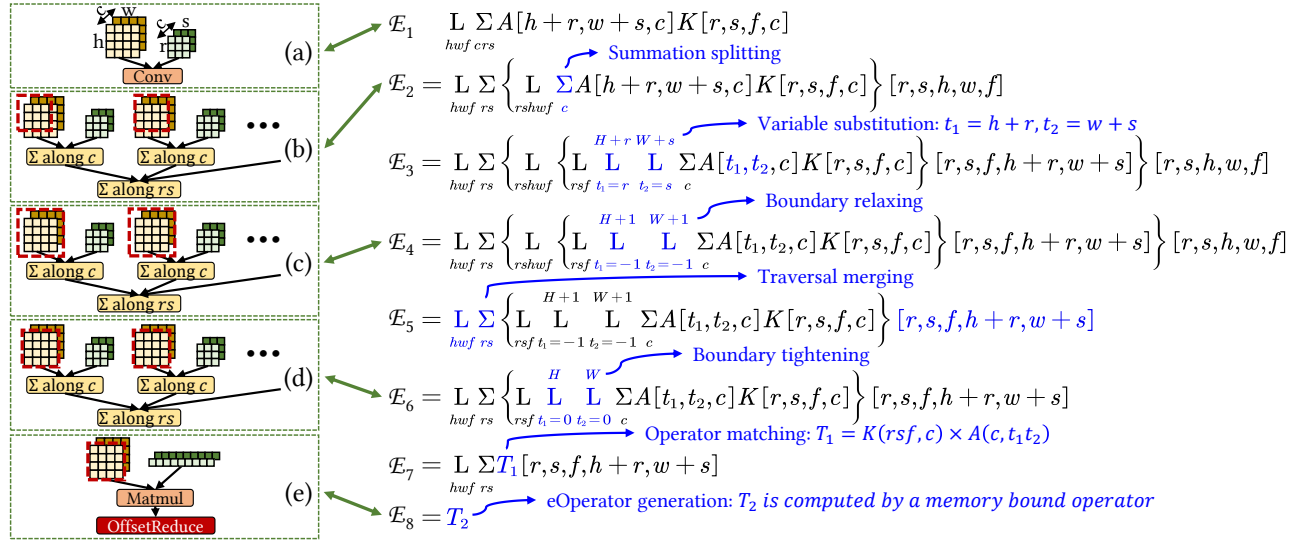


Figure 5: The derivation process of the example in Figure 3(b), which transforms Conv with Matmul and eOperators

$$\mathbf{L}_{\vec{x} \vec{s}_1 \vec{s}_2} \sum f(\mathbf{T}[\tau(\vec{x}, \vec{s}_1, \vec{s}_2)]) \Rightarrow \mathbf{L}_{\vec{x} \vec{s}_1} \left\{ \mathbf{L}_{\vec{x}\vec{s}_1 \vec{s}_2} \sum f(\mathbf{T}[\tau(\vec{x}, \vec{s}_1, \vec{s}_2)]) \right\} [\vec{x}, \vec{s}_1]$$

where  $\tau$  is a mapping from  $(\vec{x}, \vec{s}_1, \vec{s}_2)$  to an input position. EINNETH divides the iterators of a summation into two disjoint groups,  $\vec{s}_1$  and  $\vec{s}_2$ , which splits the summation into two nested scopes  $\mathcal{S}_1$  and  $\mathcal{S}_2$ , where  $\mathcal{S}_1$  is the highlighted part in the above expression and  $\mathcal{S}_2 = \mathbf{L}_{\vec{x}} \sum_{\vec{s}_1} \mathcal{S}_1[\vec{x}, \vec{s}_1]$ . Note that in summation splitting, EINNETH converts the result of the inner summation into a scope, whose output is reused by the outer summation.

To transform a  $3 \times 3$  convolution to a batch of nine matrix multiplications, as shown in Figure 5, EINNETH first transforms the initial expression  $\mathcal{E}_1$  to  $\mathcal{E}_2$  by splitting the summation  $\sum_{crs}$  into two summations  $\sum_{rs}$  and  $\sum_c$ , and instantiating the output of the inner summation (i.e.,  $\{\mathbf{L}_{rshwf} \sum_c A[h+r, w+s, c]K[r, s, f, c]\}$ ). The inner scope only sums along the  $c$  dimension; as a result, an intermediate five-dimensional tensor is instantiated since the summation along the  $r$  and  $s$  dimensions is not performed but converted to traversal notations. The outer scope computes the remaining summation over the  $r$  and  $s$  dimensions, which produces a three-dimensional tensor. Figure 5 (a) and (b) show the change in computation graph.

**Variable substitution** substitutes a set of traversal iterators  $\mathbf{L}_{\vec{x}}$  with a new set of iterators  $\mathbf{L}_{\vec{y}}$  by applying a *bijective* function  $\Phi$  (i.e.,  $\vec{y} = \Phi(\vec{x})$ ). This transformation allows the expression to be computed using a different set of traversal iterators. In particular, for an expression  $\mathbf{L}_{\vec{x}} \sum f(\mathbf{T}[\tau(\vec{x})])$ , variable substitution introduces an intermediate scope that computes  $\mathbf{L}_{\vec{y}} \sum f(\mathbf{T}[\tau(\Phi^{-1}(\vec{y}))])$ , where  $\Phi$  is a bijective function that maps the iterating space  $\mathbb{X}$  to  $\Phi(\mathbb{X})$ , and  $\Phi^{-1}$  is the reverse function of  $\Phi$ :

$$\mathbf{L}_{\vec{x}} \sum f(\mathbf{T}[\tau(\vec{x})]) \Rightarrow \mathbf{L}_{\vec{x}} \left\{ \mathbf{L}_{\vec{y}} \sum f(\mathbf{T}[\tau(\Phi^{-1}(\vec{y}))]) \right\} [\Phi(\vec{x})].$$

A variable substitution constructs an intermediate scope with new traversal iterators. To preserve functional equivalence, the original iterator  $\vec{x}$  is used to construct the final result using the output of the intermediate scope.

Although numerous possible variable substitutions exist for an expression, EINNETH infers legal ones by analyzing indexing functions in expressions and checking whether they can form bijections. In Figure 5, EINNETH applies a variable substitution to transform the expression from  $\mathcal{E}_2$  to  $\mathcal{E}_3$  using a bijective function  $\Phi$  that maps  $(r, s, f, h+r, w+s)$  to  $(r, s, f, t_1, t_2)$ . Specifically,  $h+r$  and  $w+s$  are substituted with  $t_1$  and  $t_2$  in  $\mathcal{E}_3$ . To automatically identify promising variable substitutions among all alternatives, §6.1 introduces expression distance, a novel technique for efficiently exploring the search space.

**Traversal merging** combines the traversal notations in two separate scopes into one scope using an indexing function  $\Phi$ :

$$\mathbf{L}_{\vec{x}} \sum_{\vec{y}} \sum_{\vec{z}} \{ \mathbf{L} f(\mathbf{T}[\tau(\vec{z})]) \} [\Phi(\vec{x}, \vec{y})] \Rightarrow \mathbf{L}_{\vec{x}} \sum_{\vec{y}} f(\mathbf{T}[\tau(\Phi(\vec{x}, \vec{y}))])$$

where indexing function  $\Phi$  maps the outer scope iterators  $\vec{x}, \vec{y}$  to the inner scope iterators  $\vec{z}$  and satisfies  $\Phi(\mathbb{X} \times \mathbb{Y}) \subseteq \mathbb{Z}$ .

In the example of Figure 5, EINNETH applies traversal merging to transform  $\mathcal{E}_4$  to  $\mathcal{E}_5$ . For this transformation, the outer traversal and summation notations and the inner traversal notation both include five iterators (i.e.,  $\vec{x} = (h, w, f)$ ,  $\vec{y} = (r, s)$ , and  $\vec{z} = (r, s, h, w, f)$ ). Traversal merging is applied with an identity mapping function  $\Phi$  and an indexing function  $\tau(r, s, h, w, f) = (r, s, f, h+r, w+s)$ . Traversal merging removes a scope and preserves the same computation graph.

**Boundary relaxing and tightening.** Boundary tightening inspects whether the computation for some output elements can be avoided if these elements are constants for arbitrary

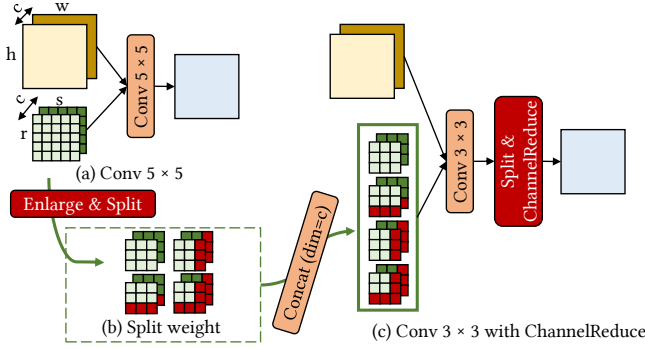


Figure 6: Conv5x5 to Conv3x3 transformation

inputs. EINNET executes constant propagation on expressions to deal with constant numbers in expressions and paddings in tensors. If an output region has constant values, EINNET converts it into an attribute of tensors to avoid unnecessary computation. In contrast, boundary relaxing enlarges tensors by adding extra paddings and redundant computations to explore more optimizations. Figure 6 shows the optimization that pads a Conv5x5 to a Conv6x6 and then converts it to a Conv3x3 with quadrupled output channels. The following formula shows how relaxing and tightening are performed:

$$\int_{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \iff \int_{\mathbb{X}'} f(\mathbf{T}[\tau(\vec{x})]),$$

where  $\mathbb{X} \subset \mathbb{X}'$ , and  $\mathbf{T}$  has a constant value in  $\mathbb{X}' \setminus \mathbb{X}$ .

To limit the number of possible candidate parameters for this rule, EINNET relaxes and tightens boundaries to a common constant. In the running example in Figure 5, the formula in  $\mathcal{E}_4$  performs boundary relaxing on  $t_1$  and  $t_2$ , transforming their ranges from  $[r, H+r)$  and  $[s, W+s)$  to  $[-1, H+1)$  and  $[-1, W+1)$ , respectively, as the ranges of  $r$  and  $s$  are  $[-1, 1]$  for a  $3 \times 3$  convolution kernel. After boundary relaxing, the computation graph is transformed from Figure 5 (b) to (c). If multiple plans exist, the most strict one is selected to prevent extra redundant computing. Meanwhile, EINNET is still able to find the transformations introducing more redundancy by applying the rule multiple times.

EINNET performs boundary tightening to transform  $\mathcal{E}_5$  into  $\mathcal{E}_6$ . In  $\mathcal{E}_5$ , as the computation performed on  $t_1 = -1$ ,  $t_1 = H$ ,  $t_2 = -1$  and  $t_2 = W$  falls in the paddings of tensor  $A$ , the computation result is zero as well. Hence, the ranges of  $t_1$  and  $t_2$  are tightened from  $[-1, H+1)$  and  $[-1, W+1)$  to  $[0, H)$  and  $[0, W)$ , respectively. After boundary tightening, the computation graph is transformed from Figure 5 (c) to (d).

**Derivation search space.** The derivation rules allow EINNET to explore a large search space of expressions. Figure 7 illustrates the derivation search space of a  $3 \times 3$  convolution. By applying different derivation rules, the initial expression is derived into various equivalent expressions, shown as the computation graphs in Figure 7. The motivating example shown in Figure 5 is identified by the derivation path

(a)  $\rightarrow$  (b)  $\rightarrow$  (c)  $\rightarrow$  (d)  $\rightarrow$  (e). The figure also shows many other expressions discovered by EINNET: By deriving the expression in (d) to Conv1x1 instead of Matmul, EINNET discovers a new expression in (f). By merging summation iterators, expression (i) adopts an eOperator to concatenate multiple inputs with offsets for the following Matmul, which represents the conventional Im2col optimization [36]. Expression (k) shows a group convolution is equivalent to the original one by duplicating its input. Expressions (n) and (p) show two additional candidate expressions, both of which decompose the  $3 \times 3$  convolution into smaller convolutions while preserving output using derived eOperators.

## 4.2 Inter-Expression Derivation

We now introduce the inter-expression derivations rule in EINNET for splitting, merging, and fusing expressions.

**Expression splitting** divides an expression into two *independent* ones by partitioning the original expression's traversal space  $\mathbb{X}$  into two subspaces  $\mathbb{X}_1$  and  $\mathbb{X}_2$ , where  $\mathbb{X} \subseteq \mathbb{X}_1 \cup \mathbb{X}_2$ :

$$\int_{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \implies \int_{\mathbb{X}_1} f(\mathbf{T}[\tau(\vec{x})]) \sim \int_{\mathbb{X}_2} f(\mathbf{T}[\tau(\vec{x})])$$

where  $\sim$  denotes the independence of the two expressions.

**Expression merging** is the reverse of expression splitting. It merges two *independent* expressions with the same computation by merging their traversal spaces  $\mathbb{X}_1 \cup \mathbb{X}_2 \subseteq \mathbb{X}$ :

$$\int_{\mathbb{X}_1} f(\mathbf{T}[\tau(\vec{x})]) \sim \int_{\mathbb{X}_2} f(\mathbf{T}[\tau(\vec{x})]) \implies \int_{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})])$$

**Expression fusion** fuses multiple *dependent* expressions into one using the following rule:

$$\int_{\mathbb{Y}} g(\mathbf{T}'[\pi(\vec{y})]) \circ \int_{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \implies \int_{\mathbb{Y}} g(\{\int_{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})])\}[\pi(\vec{y})])$$

where  $\mathbf{T}'$  is equal to the computation result of the highlighted part in the above expression, and  $\mathcal{E}_1 \circ \mathcal{E}_2$  denotes that the result of expression  $\mathcal{E}_2$  is fed as inputs to expression  $\mathcal{E}_1$ .

Figure 3(a) shows a sequence of derivations involving inter-expression derivation. EINNET first applies intra-expression derivation rules to transform Conv3x3 and Conv1x1 to two Matmuls and an eOperator. Since the two Matmuls share the same input and computation pattern, EINNET is able to apply the expression merging rule upon them. As shown in the dashed box, EINNET transposes and concatenates the two weight tensors as the input for Matmul. The outputs of Matmul are split to get two equivalent outputs. Furthermore, EINNET applies the expression fusion rule to perform vertical operator fusion, an optimization fusing a chain of operators into a single kernel to reduce data movement and kernel launch overhead. In the solid boxes in Figure 3(a), EINNET fuses memory-bound operators (e.g., OffsetReduce and Relu) into one eOperator by applying expression fusion.

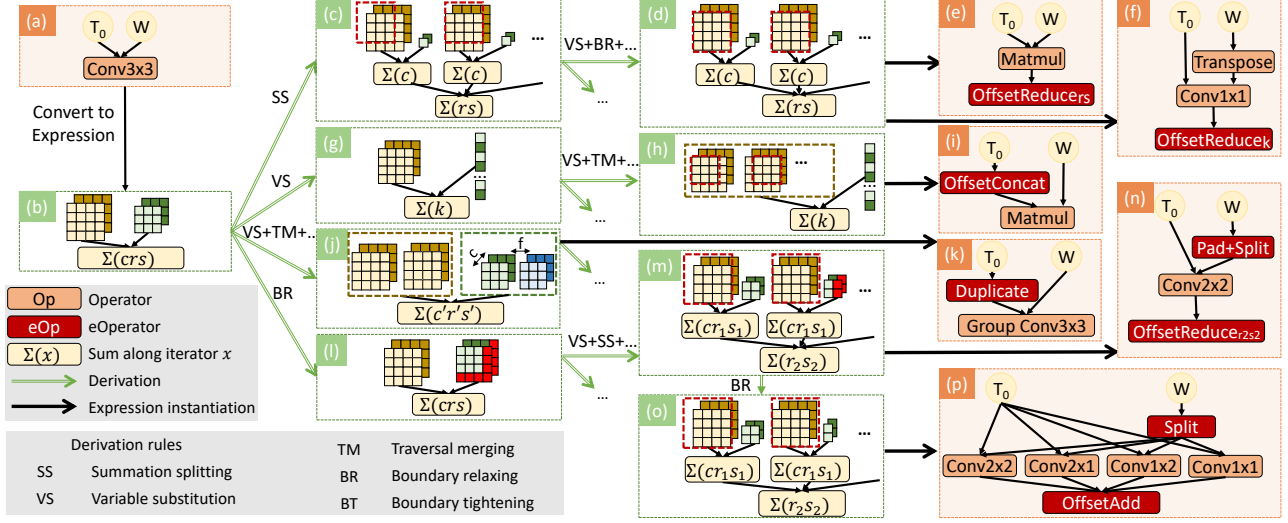


Figure 7: Derivation procedure for a subgraph of a convolution. Data layout transformation operators and intermediate derivation steps are omitted for conciseness. The output channel dimension of convolution kernel is only shown in (j) and denoted by  $f$ .

## 5 Expression Instantiation

Although EINNET can treat all expressions as eOperators and use an off-the-shelf kernel generator (e.g., TVM in our implementation) to generate executable programs, doing so would result in suboptimal performance. This is because existing vendor-provided tensor libraries such as cuDNN [10] and cuBLAS [11] include a collection of highly optimized tensor algebra kernels that outperform their counterparts generated by tensor compilers. The performance and expressiveness trade-off between hand-tuned and auto-generated kernels introduces both challenges and opportunities: we should opportunistically lower some expressions to vendor-provided kernels to realize their performance advantages and use kernel generators to generate executable programs for remaining expressions. We refer to this task as *expression instantiation*.

EINNET considers two derivation rules for expression instantiation: (1) *operator matching* allows EINNET to opportunistically use existing highly optimized kernels (e.g., cuDNN [10] and cuBLAS [11]) to achieve high performance, and (2) *eOperator generation* enables flexible kernel generation for an arbitrary eOperator. After applying these rules, the instantiated scopes are replaced with tensors in the original expression and are separated from the following derivation.

To lower expressions to kernels, EINNET uses a strategy that maps compute-intensive expressions to predefined operators and employs a kernel generator for memory-bound expressions. This strategy allows EINNET to benefit from existing vendor libraries and maintain low compilation time, since memory-bound expressions usually involve a small schedule space in existing code generation frameworks [7]. While a more aggressive utilization of kernel generators has the potential to outperform the opportunistic strategy, it introduces significant kernel tuning overhead for millions of

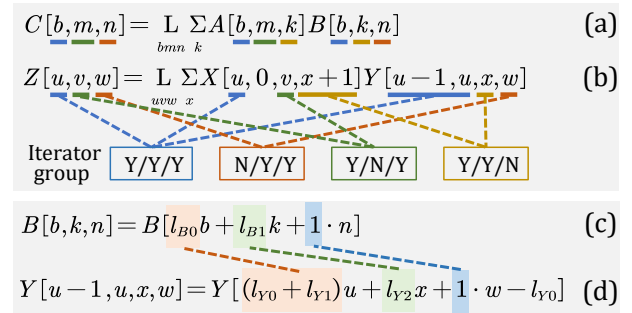


Figure 8: Match an expression to BatchMatmul. Expression (a) and (b) show iterator groups and Expression (c) and (d) show matching attributes with flattened expressions.

possible expressions during the program optimization. This is due to the difficulty in accurately estimating the performance of a kernel without actually tuning and profiling it.

To determine whether an expression is compute-intensive or memory-bound, EINNET analyzes its arithmetic intensity, calculated as the ratio between its FLOPs and tensor sizes. Expressions with arithmetic intensity lower than a threshold (4 in our evaluation) are considered memory-bound eOperators. EINNET decides whether to perform operator matching or eOperator generation for this expression based on this metric. The following introduces these two instantiation rules.

### 5.1 Operator Matching

Mapping an expression to a predefined operator is challenging since an operator can be represented in various expressions. For example, while expressions in Figure 8(a-b) have distinct forms, they can both be instantiated as batched matrix multiplication kernels in cuBLAS as it supports tensors with flexible data layouts.



Table 2: Iterator mapping table. Iterators are categorized by where they appear in expressions. For each iterator group, Y and N indicate whether the iterators appear in the index of corresponding tensors.

Operators	Tensor algebra expressions	Iterator groups ( $\mathbf{I}_0/\mathbf{I}_1/\mathbf{O}_0$ )			
		Y/Y/Y	Y/N/Y	N/Y/Y	Y/Y/N
Add	$\mathbf{O}_0[m, n] = \mathbf{L}_{mn} \mathbf{I}_0[m, n] + \mathbf{I}_1[m, n]$	$m, n$			
BatchMatmul	$\mathbf{O}_0[b, m, n] = \mathbf{L}_{bmn} \sum_k \mathbf{I}_0[b, m, k] \mathbf{I}_1[b, k, n]$	$b$	$m$	$n$	$k$
Conv	$\mathbf{O}_0[n, h, w, f] = \mathbf{L}_{nhwf} \sum_{crs} \mathbf{I}_0[n, h+r, w+s, c] \mathbf{I}_1[r, s, f, c]$		$n, h, w$	$f$	$c, r, s$
G2BMM	$\mathbf{O}_0[b, m, w] = \mathbf{L}_{bmw} \sum_k \mathbf{I}_0[b, m, k] \mathbf{I}_1[b, k, m+D*(w-W)]$	$b, m$		$w$	$k$

EINNET uses an *iterator mapping table* to determine if a given expression can be mapped to a predefined operator, where iterators of each operator are grouped based on whether the iterator appears in the operator’s input/output tensors. Table 2 shows the iterator mapping table for several operators with two input and one output tensors, including element-wise operators, batched matrix multiplication, convolution, and G2BMM [23] (general to band matrix multiplication). Each row in the table corresponds to an operator, while each column shows an *iterator group*. The iterator mapping table also records the coefficients of iterators in the index of each tensor for operator matching. It can be extended to support operators with an arbitrary number of inputs and outputs.

The iterator mapping table allows EINNET to determine if an expression can be mapped to an operator as follows:

- Match tensors.** To map a given expression to an operator, EINNET enumerates all possible one-to-one mappings between the expression and operator’s input/output tensors. For example, to map the expression in Figure 8 (b) to BMM (i.e., expression in Figure 8 (a)), there exist two possible mappings,  $\{A \rightarrow X, B \rightarrow Y\}$  and  $\{A \rightarrow Y, B \rightarrow X\}$ .
- Match iterators.** For each tensor mapping, EINNET further enumerates all possible ways to match iterators between the expression and operator using the iterator mapping table described above. For example, assuming a tensor mapping  $\{A \rightarrow X, B \rightarrow Y\}$  in Figure 8, iterators  $\{u, v, x, w\}$  in (b) are mapped to iterators  $\{b, m, k, n\}$  in (a) based on the iterator mapping table (iterators in the same group are marked in the same color). When there are multiple iterators in the same group, EINNET enumerates all possible mappings between these iterators.
- Match operator attributes.** Many predefined operators contain attributes to specify computation. E.g., modern BLAS libraries use *lda* and *ldb* to control the data layouts for input tensors in matrix multiplication. To match these attributes, EINNET flattens the input and output tensors (i.e., reshapes them into one-dimensional tensors) to hide the complexity of tensor shapes. EINNET then matches the operator attributes by examining the variable coefficients of the flattened tensors. Figure 8(c-d) show how EINNET determines the attributes  $l_{B0}$  and  $l_{B1}$  for a BMM operator. It flattens the tensor B in both expressions and compares their coefficients:  $l_{B0} = l_{Y0} + l_{Y1}$ ,  $l_{B1} = l_{Y2}$ , where  $l_{Yn}$  is the stride of  $n$ -dimension of tensor  $Y$ . The coefficient of  $w$  in Expression (d) is also checked to be equal to that of  $n$

```
T1 = tvn.te.compute((H, W, F), lambda h, w, f:
    tvn.te.sum(T1[r, s, h+r, w+s, f], axis=[r, s]))
```

Figure 9: Lowering  $\mathcal{E}_7$  in Figure 5 to TVM.

in Expression (c), as they are a pair of matched iterators.

## 5.2 eOperator Generation

For expressions that cannot be mapped to vendor-provided predefined operators, EINNET converts them into eOperators. Since an eOperator precisely defines its computation, EINNET can directly feed it to an off-the-shelf kernel generation framework (e.g., TVM [7]). For example, for expression  $\mathcal{E}_7$  in Figure 5, which corresponds to `OffsetReduce` in the transformed computation graph, EINNET feeds it to TVM by converting its iterator space into a tensor and the computation expression into a lambda function. Figure 9 shows the TVM code generated by EINNET for expression  $\mathcal{E}_7$ , which can be an input program to TVM to generate an executable kernel.

## 6 Program Optimizer

This section describes EINNET’s *program optimizer*, which uses the expression derivation and instantiation techniques described in §4 and §5 to optimize input tensor programs. These derivation rules create a large and complex search space of programs functionally equivalent to the input. EINNET uses a *distance-guided* search algorithm to explore the space (§6.1) and develops a *fingerprinting* technique to prune redundancy (§6.2). Finally, §6.3 describes how EINNET orchestrates these techniques to perform end-to-end optimizations.

### 6.1 Distance-Guided Search

To explore the search space created by EINNET’s derivations, a purely randomized search strategy can only explore either a limited set of paths or small searching depths, leading to suboptimal performance. To address this challenge, EINNET develops a two-stage *distance-guided* search algorithm to apply derivations. It includes an *explorative derivation* stage and a *converging derivation* stage, as shown in Figure 10.

**Explorative derivation.** During explorative derivation, EINNET iteratively applies *all* derivation rules to the current expression. A hyperparameter *MaxDepth* determines

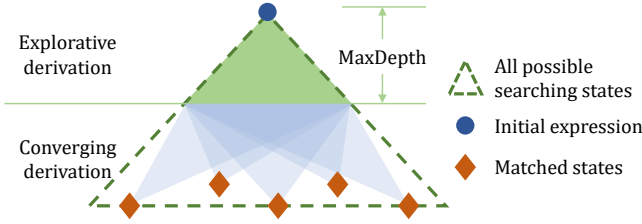


Figure 10: Distance-guided search

the maximum number of derivation rules EINN<sub>ET</sub> applies during explorative derivation. As described in §5, EINN<sub>ET</sub> opportunistically uses vendor-provided kernel libraries to maximize performance. Thus, EINN<sub>ET</sub> leverages converging derivation to quickly derive an expression toward a target operator (e.g., operators in cuDNN and cuBLAS). EINN<sub>ET</sub> automatically generates necessary eOperators to bridge the gap between the current expression and target operator.

**Converging derivation.** During converging derivation, EINN<sub>ET</sub> first selects a target operator and uses a novel metric, *expression distance*, to guide the applications of derivation rules in this stage. Expression distance measures the difference between a given expression  $\mathcal{E}_1$  and the canonical expression of a given operator  $\mathcal{E}_2$ . To calculate the distance between  $\mathcal{E}_1$  and  $\mathcal{E}_2$ , EINN<sub>ET</sub> first matches all iterators in  $\mathcal{E}_1$  and  $\mathcal{E}_2$  using the iterator mapping table (see §5.1) and counts the total number of mismatched iterators as their distance.

Specifically, each iterator mismatch between the current expression and target operator indicates that the two expressions have a different number of iterators in an iterator group (see Table 2). EINN<sub>ET</sub> applies derivation rules to fix mismatches, such as variable substitution rules to merge/split iterators, resulting in reduced expression distances. For example, to derive the expression in the inner scope of  $\mathcal{E}_6$  in Figure 5 to a Matmul, EINN<sub>ET</sub> compares their iterators (Table 2) and obtains the following matches:  $t_1, t_2 \rightarrow m; r, s, f \rightarrow n; c \rightarrow k$ . To fix mismatches, EINN<sub>ET</sub> applies variable substitutions to merge iterators  $t_1$  and  $t_2$  into  $m$  and merge  $r, s, f$  into  $n$ .

After all iterators are matched, EINN<sub>ET</sub> infers the shape of each input/output tensor according to the target operator and constructs new tensors from existing ones by adding eOperators. For example, the new input tensor  $\mathbf{A}'$  and weight tensor  $\mathbf{K}'$  for Matmul are constructed by the following expressions:

$$\mathbf{A}'[m, k] = \mathbf{A}'[t_1 \times W + t_2, c] = \mathbf{A}[t_1, t_2, c] \quad (2)$$

$$\mathbf{K}'[k, n] = \mathbf{K}'[c, r \times S \times F + s \times F + f] = \mathbf{K}[r, s, f, c], \quad (3)$$

where the mapping functions are  $(m, k) = \Phi_A(t_1, t_2, c) = (t_1 \times W + t_2, c)$  and  $(k, n) = \Phi_K(r, s, f, c) = (c, r \times S \times F + s \times F + f)$ , and  $W, S,$  and  $F$  are the range of the iterators  $w, s$  and  $f$ . EINN<sub>ET</sub> automatically generates Expression (2) and (3) to fix the mismatch and reduce the expression distance.

During converging derivation, EINN<sub>ET</sub> only considers derivations that reduce the expression distance of the current expression and target operator, allowing EINN<sub>ET</sub> to prune

most derivations and quickly converge to the target operator. By enumerating operators in the iterator mapping table as the target operator, EINN<sub>ET</sub> finds transformations involving different operators.

**Delayed code generation.** To accelerate the search, EINN<sub>ET</sub> estimates the performance of derived programs to avoid frequent code generation for eOperators. Specifically, the execution time of a predefined operator is measured by profiling its kernel on hardware. Meanwhile, the run time of an eOperator is estimated based on its input/output sizes and hardware memory bandwidth. We observe that this estimation is accurate since eOperators are memory-bound and usually account for a small part of the total execution time.

## 6.2 Redundancy Pruning

Applying different sequences of derivations may result in the same expression. For example, splitting an iterator into two and then merging them results in the original one. To prune redundancy, EINN<sub>ET</sub> uses a *fingerprint* technique to detect duplicate expressions. A *fingerprint* is a hash of an expression and can eliminate the following sources of redundancy:

- **Summation reordering:** summations can be reordered, e.g.,  $\sum_{\vec{x}} \sum_{\vec{y}} f(\vec{x}, \vec{y})$  is equivalent with  $\sum_{\vec{y}} \sum_{\vec{x}} f(\vec{x}, \vec{y})$ . Note that traversal reordering does not imply equivalence since it involves layout transformations.
- **Operand reordering:** operands of commutative binary operations can be reordered, e.g.,  $L_{\vec{x}}(\mathbf{T}_1[\vec{x}] + \mathbf{T}_2[\vec{x}])$  is equal to  $L_{\vec{x}}(\mathbf{T}_2[\vec{x}] + \mathbf{T}_1[\vec{x}])$ . Operand reordering should be applied for both iterator computation and tensor computation.
- **Iterator renaming:** iterators should be distinguished by their iterator space instead of names, e.g.,  $L_{x=0}^N L_{y=0}^M f(x, y)$  and  $L_{y=0}^N L_{z=0}^M f(y, z)$  are equivalent, and  $(x, y)$  in the former one should be mapped to  $(y, z)$  in the latter one.
- **Tensor renaming:** tensors introduced by different scopes may have the same value.

To eliminate the above sources of redundancy, EINN<sub>ET</sub> adopts the following methods to calculate fingerprints. For a traversal iterator, EINN<sub>ET</sub> uses its iterator space and its order relative to all other traversal notations in the current scope as its fingerprint. Since order is considered, fingerprint can differentiate traversal iterators with the same iterator spaces but in different locations of the traversal notations. For a summation iterator, EINN<sub>ET</sub> only uses its iterator space as its fingerprint. Thus expressions under summation reordering have the same fingerprint. To account for operand reordering, EINN<sub>ET</sub> uses the operation type and an *order-independent* hash for commutative operations (e.g., addition) and an *order-dependent* hash for other operations. The fingerprint of a tensor depends on its source. For an input tensor, EINN<sub>ET</sub> calculates its fingerprint by hashing its name. For an intermediate tensor generated by a scope, its fingerprint is identical to that of the expression that produces the tensor.

---

**Algorithm 1** Program-level optimizer.

---

```
1: Input: An input tensor program  $\mathcal{P}$ 
2: Output: An optimized tensor program  $\mathcal{P}_{\text{opt}}$ 
3:
4:  $I\mathcal{R}$  = inter-expression rule set
5:  $\mathcal{S}\mathcal{P}$  = split  $\mathcal{P}$  and translate subprograms into expressions
6:  $\mathcal{P}_{\text{opt}} = \emptyset$ 
7: for  $E_0 \in \mathcal{S}\mathcal{P}$  do
8:    $Q = \{E_0\}$ 
9:   for  $E \in Q$  do
10:    for  $r \in I\mathcal{R}$  do
11:       $Q = Q + r(E)$ 
12:     $Q = Q + \text{DISTANCEGUIDEDSEARCH}(E)$ 
13:    Add the best transformation in  $Q$  into  $\mathcal{P}_{\text{opt}}$ 
14: POSTOPTIMIZATION( $\mathcal{P}_{\text{opt}}$ )
15: return  $\mathcal{P}_{\text{opt}}$ 
```

---

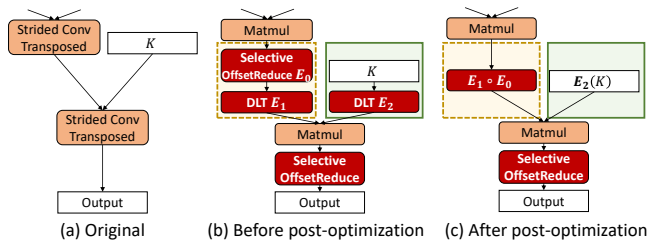


Figure 11: Post-optimization for InfoGAN. Red blocks represent eOperators. DLT means data layout transformation.

### 6.3 End-to-End Workflow

Algorithm 1 shows EINN<sub>ET</sub>'s workflow for optimizing an input tensor program in an end-to-end fashion. For an input program, EINN<sub>ET</sub> first splits it into multiple subprograms using non-linear activation operators as the splitting points. This is because activation operators often do not provide further optimization opportunities other than fusion, as discovered by prior work [38]. For each subprogram, EINN<sub>ET</sub> translates it into expressions using the canonical expression of each operator. Since a subprogram may include multiple operators and thus multiple expressions, EINN<sub>ET</sub> applies inter-expression derivation rules (Line 11) and feeds each expression to the distance-guided search (§6.1) for performing *intra-expression* derivations (Line 12). Instead of integrating intra- and inter-expression optimizations in a unified search space and performing them jointly, the separate search prioritizes the transformations that can map expressions into operators. Thus, EINN<sub>ET</sub> is able to find promising transformations quickly and prune unnecessary search states according to the execution time of transformed results.

Finally, EINN<sub>ET</sub> selects the best-discovered expression of each subprogram, performs post-optimization, and generates an optimized tensor program. Figure 11 shows two types of post-optimization: eOperator fusion and compile-time expression evaluation. EINN<sub>ET</sub> generates eOperators to facilitate optimizing transformations when optimizing a subprogram. During post-optimization, consecutive eOperators are fused into a single eOperator by applying inter-expression

derivations. The dashed boxes in Figure 11(b) and (c) show such cases. EINN<sub>ET</sub> also detects compile-time computable expressions to reduce runtime overhead. For example, the data layout transformation  $E_2$  in Figure 11 can be processed during post-optimization.

## 7 Evaluation

### 7.1 Experimental Setup

**Implementation of EINN<sub>ET</sub>.** EINN<sub>ET</sub> is built with over 23K lines of C++ and Python code. We realize the tensor expression derivation system from scratch and implement an execution runtime for tensor programs. Users can both define tensor programs in EINN<sub>ET</sub> directly and load existing ones in the ONNX format [29]. To support an operator in derivation, EINN<sub>ET</sub> requires its tensor expression and operator attribute constraints to automatically convert it between expressions and operators. We set the default maximum search depth of explorative derivation to 7, which is an empirical configuration satisfying both optimization quality and search time.

**Platform.** We evaluate EINN<sub>ET</sub> on a server with dual Intel Xeon E5-2680 v4 CPUs, NVIDIA A100 40GB and V100 32GB PCIe GPUs. All experiments use CUDA 11.0.2, cuBLAS 11.1.0, and cuDNN 8.0.3.

**Workloads.** We evaluate EINN<sub>ET</sub> on seven DNN models, spanning various fields and covering both classic and emerging DNNs. InfoGAN [9] is a generative adversarial network learning disentangled representations from objects. DCGAN [32] leverages deep convolution structures to get hierarchical representations. FSRCNN [13] is used for fast image super-resolution. GCN [30] is an image semantic segmentation model. ResNet-18 [19] is a famous image classification network. CSRNet [25] adopts dilated convolution for congested scene analysis. LongFormer [6] is an improved Transformer model dealing with long-sequence language processing. We adopt typical input shapes based on the papers and implementations of models to keep the evaluation meaningful in real scenarios.

### 7.2 End-to-End Performance

We first compare the end-to-end inference performance of EINN<sub>ET</sub> against today's DNN frameworks, including TensorFlow v2.4 [4], TensorFlow XLA [2], Nimble [22], TVM v0.10 with Ansor [7], TensorRT v8.2 [35], and PET v0.1 [38]. All frameworks use the same version of cuBLAS and cuDNN as their backend and the same data type FP32 in computation for a fair comparison. For the new attention operator in Longformer, we provide an auto-tuned kernel for TVM, TensorRT, PET, and EINN<sub>ET</sub>, and implement it by

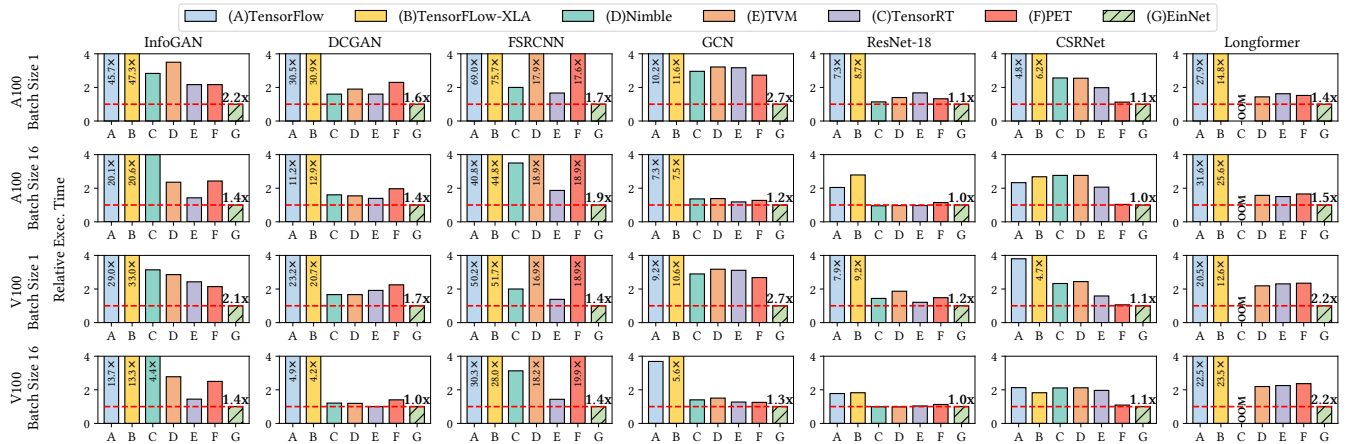


Figure 12: End-to-end performance comparison with other systems on an A100 and a V100 GPU with batch sizes of 1 and 16. OOM means out of memory. Bars over 4× are truncated, and their relative execution times to EINNET are marked on the bars. The numbers above EINNET’s bars show EINNET’s speedups over the best baseline.

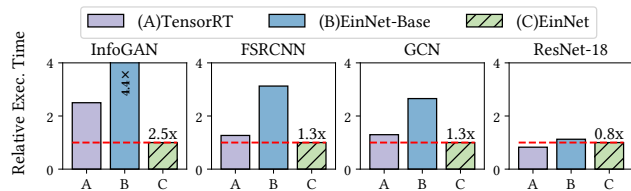


Figure 13: End-to-end performance comparison with TensorRT on an A100 with TF32 and batch sizes of 1. The numbers above EINNET’s bars show EINNET’s speedups over the best baseline.

einsum in other frameworks. Figure 12 shows the results on NVIDIA A100 and V100 GPUs under batch sizes 1 and 16.

EINNET outperforms the best existing baseline by up to 2.72× on A100 and 2.68× on V100. For both CNNs (e.g., GCN) and language models (e.g., Longformer), EINNET is able to improve their performance by more than 2×. Among the seven models, ResNet-18 has been heavily optimized by existing tensor program frameworks and optimizers; however, EINNET still outperforms existing optimizers by 1.2× on V100, by applying the novel transformations shown in Figure 3. For CSRNet, a typical optimization case of PET, EINNET discovers similar transformations by derivations and eliminates extra introduced transposes, indicating that EINNET’s derivation rules can perform PET’s optimizations and uncover additional improvements.

Figure 13 shows the speedup with the computation data type of TF32 and Tensor Cores on A100. To show the benefits provided by EINNET, we create a baseline EINNET-Base which executes models in EINNET with derivation optimizations disabled. As shown in Figure 13, while EINNET usually brings significant speedups over EINNET-Base and TensorRT, TensorRT can have better performance in models like ResNet-18. Though TensorRT is not open source, the profiling results

Table 3: Performance studies on the cases in §7.3. The Algo column shows the best convolution algorithm for cuDNN, where IGEMM, FFT, and WINO refer to implicit GEMM, Fast Fourier Transform, and Winograd [24] algorithms. The DRAM and L2 columns show the amount of memory access.

	Input shape	Conv Algo	Time (ms)	DRAM (MB)	L2 (MB)	
Conv3x3 Figure 3 (b)	[1,512,7,7]	Original	WINO	0.126	56.7	70.6
		Optimized	N/A	0.046	10.5	27.5
Conv-Transpose Figure 6	[16,448,2,2]	Original	IGEMM	0.136	7.74	122
		Optimized	N/A	0.032	8.07	27.8
Conv5x5 Figure 6	[16,32,224,224]	Original	FFT	0.854	547	579
		Optimized	WINO	0.528	368	499
G2BMM Figure 14	[8,10000,64]	Original	N/A	7.14	20.9	19750
		Optimized	N/A	1.57	20.6	817

show that it leverages many efficient GPU kernels besides cuBLAS and cuDNN. This can be an important source of its high performance, which is beyond the current space of EINNET.

### 7.3 Optimization Analysis

This section analyzes the optimizations discovered by EINNET on these DNNs. To highlight transformations beyond the scope of existing tensor program optimizers, we focus on transformations involving eOperators.

**Transforming operator types.** EINNET is able to opportunistically substitute an inefficient operator with well-optimized operators of different types. In ResNet-18 and InfoGAN, the transformations from Conv and ConvTranspose to Matmul are profitable. Table 3 shows a detailed performance analysis. As shown in Figure 3(b), EINNET transforms a Conv3x3 to a Matmul and an eOperator (OffsetReduce), which significantly reduces GPU DRAM accesses from 56.7 MB to 10.5 MB and achieves a 2.7× speedup. As another



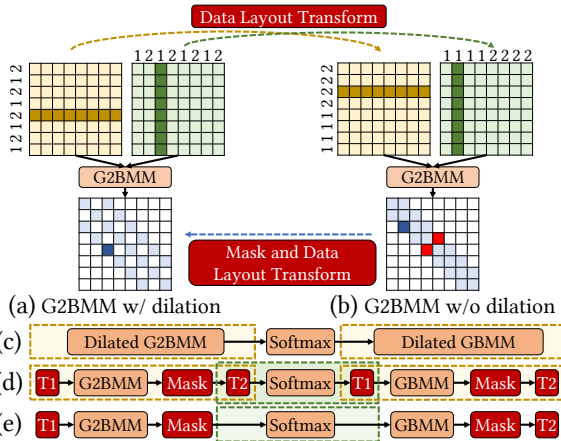


Figure 14: Optimization for Longformer Attention block.  $T_1$  and  $T_2$  are two reciprocal data layout transformations.

example, EINNET also derives a strided `ConvTranspose` to a `Matmul` and another eOperator that selectively aggregates the output of `Matmul` according to the derived expression. This transformation significantly reduces L2 access, a key contribution to performance optimization.

**Transforming operator attributes.** EINNET can also transform operator attributes by leveraging eOperators. Figure 6 shows such an optimization for convolution, which converts its kernel size from  $5 \times 5$  to  $3 \times 3$ , allowing EINNET to use more advanced convolution algorithms best suited for  $3 \times 3$  convolutions. To realize this transformation, an eOperator is added to split the output of `Conv3x3` across the channel dimension and reduce the intermediate results with corresponding offsets. Although padding the convolution kernel introduces additional computation, Table 3 shows it enables using the Winograd algorithm for convolution, which further reduces compute time and memory access.

**Transforming tensor layouts.** eOperators allow EINNET to accelerate DNN computation by optimizing tensor layouts. Figure 14 shows such a layout optimization for Longformer, which uses a dilated G2BMM (general to band matrix multiplication) to compute sparse attention. G2BMM has the same computation pattern as `Matmul` and only computes a subset of output. The blue boxes in Figure 14(a) show the output locations with a dilation of 2. EINNET discovers an optimizing layout transformation that reorders the odd and even rows or columns, converting the dilated G2BMM to a non-dilated one, as shown in Figure 14(b), which greatly reduces non-contiguous memory accesses at the cost of introducing two redundant elements (marked as red in the figure). As shown in Table 3, this transformation can reduce L2 cache access by 95.9% and execution time by 78.0%.

**Transforming graph structures.** For the Longformer case shown in Figure 14(d), four data layout transformations are

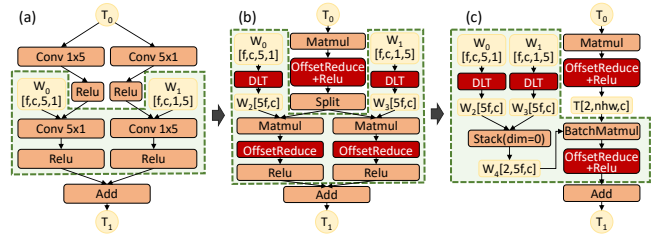


Figure 15: Optimization for the spatial separable convolutions in GCN.  $c$  and  $f$  are the numbers of input and output channels.

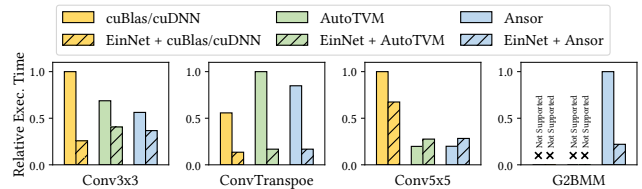


Figure 16: Operator performance before and after optimization (opt) on the math libraries and code generation framework Anso. The input settings are shown in Table 3.

introduced to accelerate dilated G2BMM. While they are not predefined operators, EINNET finds that the middle two are reciprocal through expression fusion and eliminates them since they do not affect the `Softmax` computation in between.

A more complex example is in GCN, which has a repeated structure of spatially separable convolutions (i.e., sequential `Conv1xKs` and `ConvKx1s`). As shown in Figure 15(b), EINNET first transforms convolutions to `Matmul`s and eOperators, and then merges the first two `Matmul`s into a single `Matmul`. While the two following `Matmul`s do not share common inputs, they have an identical computation pattern and can be merged into a `BatchMatmul` by applying the expression merging and operator matching rules. Note that the left part of Figure 15(c) is computed at compile-time by EINNET since it only involves weight tensors. These transformations optimize subgraph execution time by  $4.9\times$  on A100 with batch size of one.

## 7.4 Integration with Different Backends

Since EINNET searches expression space, it can cooperate with different backends, including math libraries and schedule-based code generation frameworks. To show the improvements of EINNET on these backends, we evaluate EINNET with cuBLAS/cuDNN, AutoTVM [7], and TVM auto-scheduler Anso [8]. The evaluation is carried out on the same transformations illustrated in §7.3.

Figure 16 shows EINNET can optimize tensor programs on different backends. For the `Conv3x3` in ResNet-18 and the `ConvTranspose` in InfoGAN, transforming them to `Matmul`s and eOperators has significant speedup over all three platforms. While the transformation from `Conv5x5` to `Conv3x3` is beneficial for cuDNN, it does not have perfor-

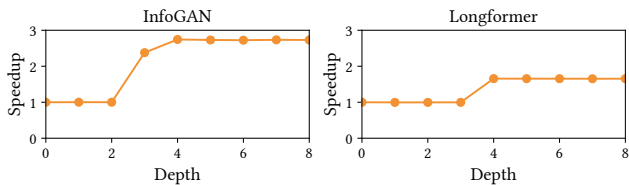


Figure 17: Speedup under different maximum search depths

performance improvement on AutoTVM and Ansoer even if efficient algorithms such as Winograd [24] are adopted. This result shows while many transformations are effective on different backends, customizing transformations for each backend is beneficial. For the G2BMM operator in Longformer, which is not a predefined operator in current math libraries, the transformation on Ansoer brings a speedup of  $4.54\times$  since less non-contiguous memory access happens.

## 7.5 Analysis of Automated Derivation

The search space of EINNET is determined by heuristic parameters, e.g., the maximum search depth for the distance-guided search algorithm (§6.1), which specifies the largest steps of derivation applied to an expression. A larger search depth enables more potential optimizations but introduces larger searching overhead. Figure 17 analyzes the speedup EINNET can achieve with different maximum search depths on InfoGAN and Longformer. On InfoGAN, EINNET has improvement when the searching step increases from 2 to 4, as new transformations are explored with a deeper search. While for Longformer, the major speedup comes from the transformation found in a 4-step derivation. In conclusion, the key takeaway is that EINNET can achieve most of the performance improvement at moderate depth.

To evaluate the proposed techniques for derivation, we evaluate the searching process on the four cases in Table 3 with and without converging derivation and expression fingerprint.

**Distance-guided derivation (§6.1)** provides a deterministic derivation direction to reduce search overhead. As shown in Figure 18(a), the search time grows exponentially with the maximum depth of explorative derivation (i.e., *MaxDepth* in Figure 10). EINNET adopts converging derivation to reduce the search depth of explorative derivation. Figure 18(b) shows the number of applied explorative derivations in these cases.

In the case of ConvTranspose, the explorative derivation requires a search with *MaxDepth* = 12 to discover the target expression. But with converging derivation, EINNET only requires a search with *MaxDepth* = 6, which means that matching a vendor-provided operator needs a six-step (12 – 6) search and converging derivation can reduce this unnecessary search. Thus, this optimization leads to a significant reduction of the search time by more than 99.0%.

**Expression fingerprint (§6.2)** prunes redundant searching states. Figure 19 shows the intermediate states and search time with and without the fingerprint mechanism. During

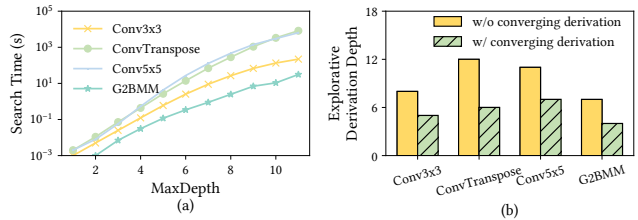


Figure 18: (a) Search time with different *MaxDepth*. (b) The number of explorative derivation steps with and without converging derivation.

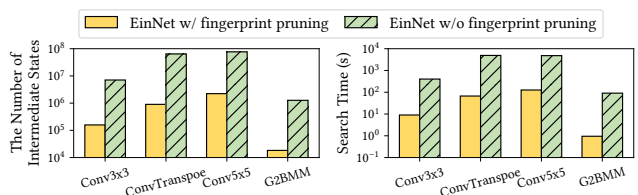


Figure 19: Ablation study of expression fingerprint pruning

the derivation, fingerprint effectively prunes 98.0% of intermediate states by recognizing and eliminating duplicate expressions and reduces 98.2% of search time on average.

With the distance-guided derivation and expression fingerprint, EINNET finishes searching within two minutes for most subprograms and is on par with existing frameworks like TASO and PET. The search spends no more than two hours for most models, which depends on the number of operators contained in models. EINNET is able to be deployed in real production environment since the search cost is one-off for a model and brings persistent benefits.

## 8 Related Work

**Rule-based approaches** such as TensorFlow XLA [2], TensorRT [35], and Grappler [1], are widely used and perform optimizations like constant folding and layout optimization. While they can efficiently optimize computation graphs, it requires extensive expert efforts and only performs manually discovered optimizations. For operator fusion, DNNFusion [28] adopts operator-level mathematical-property-based graph rewrite rules, such as associative and commutative properties. However, such rewriting rules are mainly designed for element-wise operators and cannot be easily extended to arbitrary operators since many complex operators, such as convolution and matrix multiplication, do not follow associative and commutative properties. EINNET derives tensor programs at expression level to exploit general program transformations, including splitting, fusing, and reorganizing computation into operators and eOperators. This avoids manually summarizing rules for each operator.

**Superoptimization-based approaches.** TASO [20] and

PET [38] use superoptimization techniques to generate graph transformations for a given set of operators. TASO adopts formal verification techniques to assure the equivalence of transformations. PET further introduces partially-equivalent transformations and correction mechanism to find more optimizations. Compared with these frameworks, EINNET extends the search space from POR transformations to general expressions by tensor algebra expression derivation.

**Schedule-based approaches.** Halide [24] decouples a program into computation and schedule and performs schedule space transformations. This idea is widely adopted by code generation frameworks, including TVM [7], FlexTensor [42], and Ansor [40]. Orthogonal to schedule-based optimizers, EINNET focuses on high-level graph transformation space and designs the architecture-independent expression derivation rules to reorganize computation into efficient operators.

**Task-based approaches.** Task, an abstraction of computation and memory operation workload, is introduced into tensor programs recently to optimize their performance. Rammer [27] divides operators into fine-grained tasks and proposes a sub-operator-level task scheduling method to exploit both intra- and inter-operator parallelism. Hidet [12] leverages task mapping in kernel generation to explore more aggressive optimizations. EINNET is compatible with these optimizers by utilizing them as execution and kernel generator backend.

**Tensorization.** TensorIR [14], AMOS [41], and Glenside [34] aim to generate tensorized code on tensor accelerators. While computation mapping is stressed in their workflows, these works focus on generating performant native code leveraging special circuits, such as fixed-shape matrix multipliers Intel AMX and NVIDIA TensorCore. In contrast, EINNET adopts expression matching to match arbitrary linear tensor algebra expressions, which are more flexible and contain undetermined parameters in the pattern expressions.

## 9 Conclusion

We propose EINNET, a derivation-based tensor program optimizer, which extends the optimization space of tensor programs from predefined operator representable transformations to general expressions and can create new operators desired by transformations. EINNET can outperform state-of-the-art frameworks by up to  $2.72\times$  on NVIDIA GPUs.

## Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Dr. Lidong Zhou, for their valuable comments and suggestions. This work is supported by National Key R&D Program of China under Grant 2021ZD0110202, NSFC

for Distinguished Young Scholar (62225206), the Young Scientists Fund of the National Natural Science Foundation of China (62202259), and China Postdoctoral Science Foundation (2022M711810). Haojie Wang is supported by the Shuimu Tsinghua Scholar Program. Jidong Zhai is the corresponding author of this paper ([zhaijidong@tsinghua.edu.cn](mailto:zhaijidong@tsinghua.edu.cn)).

## References

- [1] Tensorflow graph optimization with grappler; tensorflow core.
- [2] Xla: Optimizing compiler for tensorflow. <https://www.tensorflow.org/xla>, 2017.
- [3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [5] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.
- [6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
- [7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors,

*13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

- [8] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS’18. 2018.
- [9] Xi Chen, Yan Duan, Rein Houthoofd, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 2180–2188, 2016.
- [10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.
- [11] Dense Linear Algebra on GPUs. <https://developer.nvidia.com/cublas>, 2016.
- [12] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 370–384, 2023.
- [13] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *European conference on computer vision*, pages 391–407. Springer, 2016.
- [14] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. *arXiv preprint arXiv:2207.04296*, 2022.
- [15] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.
- [16] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.
- [17] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.
- [18] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2016.
- [20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.
- [21] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.
- [22] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.
- [23] Johannes Langer. *Band Matrices in Recurrent Neural Networks for Long Memory Tasks*. PhD thesis, 2018.
- [24] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.
- [25] Yuhong Li, Xiaofan Zhang, and Deming Chen. Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1091–1100, 2018.
- [26] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.
- [27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 881–897, 2020.



- [28] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.
- [29] Onnx. <https://onnx.ai/>, 2019.
- [30] Chao Peng, Xiangyu Zhang, Gang Yu, Guiming Luo, and Jian Sun. Large kernel matters—improve semantic segmentation by global convolutional network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4353–4361, 2017.
- [31] Tensors and Dynamic neural networks in Python with strong GPU acceleration. <https://pytorch.org>, 2017.
- [32] Alec Radford, Luke Metz, and Soumith Chintala. Un-supervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.
- [33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.
- [34] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2021, page 21–31, New York, NY, USA, 2021. Association for Computing Machinery.
- [35] NVIDIA TensorRT: Programmable inference accelerator. <https://developer.nvidia.com/tensorrt>, 2017.
- [36] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication, 2017.
- [37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [38] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.
- [39] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [40] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.
- [41] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA*, pages 874–887, 2022.
- [42] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

## A Artifact Appendix

### Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the paper: EINNET: Optimizing Tensor Programs with Derivation-Based Transformations.

### Scope

This artifact can be used for evaluating and reproducing the main results of the paper, including the model-level evaluation, operator-level evaluation, and the ablation studies and hyperparameter studies on search strategies.

### Contents

The following evaluation results are contained in the artifact:

**E1:** An end-to-end performance comparison between EinNet and other frameworks. (Figure 12)

**E2:** Performance studies on the cases in §7.3. (Table 3)

**E3:** Operator performance before and after optimization on the math libraries and code generation framework Anso. (Figure 15)

**E4:** Speedup under different maximum search depths. (Figure 16)

**E5:** Search time with different MaxDepth and the number of explorative derivation steps with and without converging derivation. (Figure 17)

**E6:** Ablation study of expression fingerprint pruning. (Figure 18)

### Hosting

The source code of this artifact can be found on GitHub: <https://github.com/zhengly123/OSDI23-EinNet-AE>, the main branch, with the commit ID 26a47d9.

### Requirements

#### Hardware dependencies

Dual Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, NVIDIA A100-PCI-40GB GPU, NVIDIA V100-PCIE-32GB

GPU.

#### Software dependencies

The artifact is evaluated on Ubuntu 22.04 LTS (Linux kernel 5.15.0-58). The artifact relies on CUDA 11.0.2 and cuDNN 8.0.3. The following frameworks are required as baselines:

1. TensorFlow 2.4
2. TensorRT 8.0
3. PET 1.0
4. Nimble with the commit ID bac6d10
5. TVM v0.10.0

#### Experiments workflow

The installation instruction and the following experiments are included in this artifact. All DNN benchmarks use synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU.

#### End-to-end performance (E1)

This experiment reproduces Figure 12 in the paper. Refer to OSDI23-EinNet-AE/0\_model/README.md to prepare the environment and data. The detailed commands for each baseline are provided in separate run.sh and readme files in subdirectories.

#### Performance studies on the cases in §7.3 (E2)

See README.md and run.sh in OSDI23-EinNet-AE/1\_op.

#### Operator performance (E3)

See README.md and run.sh in OSDI23-EinNet-AE/2\_kernel\_generator.

#### Speedup & Depth (E4)

See README.md and evaluate\_max\_depth.py in OSDI23-EinNet-AE/3\_search\_depth.

#### Search Time (E5)

See README.md and run.sh in OSDI23-EinNet-AE/4\_search\_time.

#### Ablation Study (E6)

See README.md and run.sh in OSDI23-EinNet-AE/5\_fingerprint.





# Hydro: Surrogate-based Hyperparameter Tuning Service in Datacenters

Qinghao Hu<sup>1,2,3</sup> Zhisheng Ye<sup>3,4</sup> Meng Zhang<sup>1,2,3</sup> Qiaoling Chen<sup>3,5</sup>  
Peng Sun<sup>3,6</sup> Yonggang Wen<sup>1</sup> Tianwei Zhang<sup>1</sup>

<sup>1</sup>Nanyang Technological University <sup>2</sup>S-Lab, NTU <sup>3</sup>Shanghai AI Laboratory  
<sup>4</sup>Peking University <sup>5</sup>National University of Singapore <sup>6</sup>SenseTime Research

## Abstract

Hyperparameter tuning is an essential step in deep learning model development that provides better model performance at the cost of substantial resources. While existing systems can improve tuning efficiency, they still fail to handle large models with billions of parameters and efficiently leverage cluster resources. Motivated by these deficiencies, we introduce Hydro, a surrogate-based hyperparameter tuning service that optimizes tuning workloads in both the job-level and cluster-level granularities. Specifically, it consists of two key components: (1) *Hydro Tuner* automatically generates and optimizes surrogate models via scaling, parametrization and fusion; (2) *Hydro Coordinator* improves tuning efficiency and cluster-wide resource utilization by adaptively leveraging ephemeral and heterogeneous resources. Our comprehensive experiments on two tuning algorithms across six models show that Hydro Tuner can dramatically reduce tuning makespan by up to  $78.5\times$  compared with Ray Tune and no reduction in tuning quality. Hydro’s source code is publicly available at <https://github.com/S-Lab-System-Group/Hydro>.

## 1 Introduction

Over the years, we have witnessed the incredible performance and rapid popularity of Deep Learning (DL) across many domains, such as vision and speech. However, it is non-trivial to acquire a qualified DL model because its performance is highly sensitive to the *hyperparameters*, which control the training process and require to be set before training [71]. Poor hyperparameters result in training instability and inferior model quality. Conversely, well-tuned hyperparameters can significantly improve model performance. For instance, PyTorch [91] recently applies a new hyperparameter recipe on ResNet-50 [41] and achieves 80.9% ImageNet classification accuracy [18], which is 4.8% higher than the former version (76.1%). Besides, RoBERTa [75] also demonstrates the critical impact of hyperparameters on the performance of large language models. Accordingly, hyperparameter tuning becomes a common practice during DL model development.

Due to the high dimensionality of the search space, a hyperparameter tuning job typically contains a large group of *trials*, each with a unique configuration [125]. To accelerate the tuning process, tech companies and researchers build hyperparameter tuning systems as cloud services [1, 8, 39, 92]

or standalone frameworks [32, 71, 72, 82, 125, 127] (Table 1). However, we argue that state-of-the-art tuning systems are still expensive and inefficient in practice, as they suffer from several fundamental problems:

- **Unacceptable cost of tuning large models.** The extraordinary performance of large foundation models (e.g., BERT [30], GPT-3 [24]) attracts wide downstream applications [3, 4, 6]. Meanwhile, the hyperparameter tuning demand for these models increases rapidly. However, all of the existing tuning systems require training multiple trials using several times of resources, which is unaffordable for large models with billions of parameters. For example, training a SOTA language model PaLM [27] of Google takes over 6,000 TPU-v4 [59] for around 2 months. Performing a hyperparameter sweep on such model is intractable [23]. Consequently, hyperparameters of most large models are not well-tuned and can lead to subpar performance [75].
- **Inefficient hardware utilization.** Recent scheduling works [46, 114, 115, 124] report that GPUs are commonly underutilized in DL clusters due to massive training jobs involving mid- or small-scale models. Moreover, despite the growing trend of foundation models being employed in clusters, large-scale models often fail to fully utilize hardware resources due to the huge communication overhead and the presence of bubbles in the pipeline parallelism [106]. To improve resource utilization, some novel tuning systems incorporate features such as elastic training [32, 71, 82], GPU sharing [125], and inter-trial fusion [110]. However, these systems have certain limitations (§8) and often require substantial resources to explore trivial trials, which results in limited resources being contributed to the final model.
- **Agnostic to cluster-wide resources.** Hyperparameter tuning jobs are pervasive and occupy enormous resources in GPU clusters. As reported by Microsoft [50, 78], “approximately 90% of models require hyperparameter tuning, with each tuning job containing 75 trials in median.” However, existing tuning systems only manage trials over the requested resources and lack interaction with cluster schedulers. Meanwhile, DL schedulers [36, 40, 46, 87, 94, 114, 123] also overlook the distinct characteristic of gradually diminishing hardware demand inherent in hyperparameter tuning jobs [71]. Consequently, the cluster encounters imbalanced resource problem: the active tuning jobs consistently occupy static resources,



leaving some of them vacant, while the queued jobs are unable to request these idle resources from the scheduler. This leads to severe queuing delay, which is exacerbated when long-term large-scale model training jobs coexist and they occupy the majority of cluster resources.

To bridge these gaps, we design Hydro, a surrogate-based hyperparameter tuning service that optimizes tuning jobs in both the job-level and cluster-level granularities via automated model scaling, fusion and interleaving. The core design of Hydro derives from the following three insights. First, *it is feasible to search hyperparameters with a smaller model*. Instead of tuning hyperparameters directly on the target model, we find it is possible to tune a model with a much smaller surrogate model by applying a novel hyperparameter transfer theory [117, 121]. Second, *cross-model fusion can be used to improve resource utilization*. Since the scaled surrogate model is prone to incur GPU underutilization, we can utilize the model architecture consistency of different trials to fuse them into a single one, achieving much higher GPU utilization and training throughput. Third, *ephemeral bubble resources in the datacenter can be leveraged for tuning*. Large model training jobs exist in the long term and occupy the majority of resources, which incurs the starvation of other jobs. We can leverage pipeline bubbles of large models to greatly extend the tuning job resources in an interleaving execution way, without hurting the training throughput of large models.

Incorporating the above insights, we build Hydro service to minimize the makespan of tuning workloads and improve the cluster-wide resource utilization. It consists of two key system components: (1) **Hydro Tuner** is the user interface that automatically generates surrogate models by *scaling* and *parametrization*. It optimizes tuning efficiency via *inter-trial* and *intra-trial fusion*, which involve combining multiple models into a single entity and subsequently performing compiler-based optimization. Besides, it efficiently orchestrates the tuning process with *adaptive fusion* and *eager transfer* mechanisms. (2) **Hydro Coordinator** is the datacenter interface that interacts with the scheduler to dynamically allocate resources and execute trials. It extends tuning resources by *interleaving training* with pipeline-enabled large model training tasks, effectively utilizing idle time intervals on each node known as *bubbles*, which are caused by the gaps between the forward and backward processing of microbatches [106]. Besides, it improves resource utilization and cluster-wide performance by *heterogeneity-aware* allocation.

To extensively assess the performance of Hydro, we conduct evaluations across 6 models, such as GPT-3 XL [24] and ResNet [41]. Experiments on Hydro Tuner show that it substantially outperforms Ray by 8.7~78.5× on makespan reduction with single-fidelity tuning algorithm, while obtaining better final model quality. Besides, our experiments on Hydro Coordinator demonstrate that interleaving with a large pipelined model can further extend the resource of tuning workload, without sacrificing the throughput of the large model.

Features	Cloud Services		HPO Frameworks		Hydro
	Vizier	SageMaker	NNI	Ray	
Distributed Environment	✓	✓	✓	✓	✓
Elastic Training	◆	◆	◆	✓	✓
Auto Model Scaling	✗	✗	✗	✗	✓
Surrogate HP Transfer	✗	✗	✗	✗	✓
Inter-Trial Fusion	✗	✗	◆	✗	✓
Intra-Trial Fusion	✗	✗	✗	✗	✓
Heterogeneity Awareness	✗	✗	✗	✗	✓
Interleaving Training	✗	✗	✗	✗	✓

Table 1: Comparison between Hydro and existing popular HPO systems: Google Vizier [39, 105], Amazon SageMaker [28, 92], Microsoft NNI [9, 127] and Anyscale Ray [72, 84]. ◆ denotes system cannot support the feature for many cases.

Table 1 compares Hydro with existing tuning systems. To summarize, we make the following contributions:

- ★ We build a holistic system that automatically applies the novel hyperparameter transfer theory together with multiple system techniques to jointly improve the tuning efficiency.
- ★ We identify the opportunities for cluster-wide optimization in the datacenter, including squeezing bubble resources with interleaving and heterogeneity-aware trial allocation.
- ★ We demonstrate the excellent performance of surrogate-based hyperparameter tuning across general models.

## 2 Background and Motivation

### 2.1 Hyperparameter Tuning

*Hyperparameter Tuning* (i.e., Hyperparameter Optimization, HPO) aims to identify the optimal hyperparameters via massive configuration exploration [71, 82]. In the general workflow of an HPO job: (1) the user designates a *search space* of hyperparameters to explore; (2) the tuning algorithm creates a set of training *trials* and each trial contains one unique hyperparameter configuration sampled from the search space; (3) the HPO system coordinates trials execution until the best hyperparameter configuration is found.

Existing research works typically optimize HPO efficiency from the tuning algorithm [33, 47, 63, 64, 67, 68, 70, 79, 104] or system [32, 60, 69, 71, 82, 110, 125, 127] aspects:

**Algorithm taxonomy.** Depending on whether to enable early stopping, tuning algorithms can be divided into two categories [100]: (1) *single-fidelity* (e.g. Random [22], Bayes [104]) algorithms require each trial to be fully trained, which is accurate but inefficient; (2) *multi-fidelity* (e.g., ASHA [63], BOHB [33]) algorithms stop unpromising trials via successive halving [53] or curve fitting [31] strategies. They are efficient but may miss the best hyperparameter configuration due to the use of “low-fidelity” evaluations. Hydro well supports both the single- and multi-fidelity algorithms.

**System optimization.** To further improve the tuning efficiency and resource utilization, there are two advanced techniques applied in state-of-the-art HPO systems: (1) *elastic training* dynamically allocates more GPU resources to the top

performing trials [71] and further adjusts the entire requested resources [32, 82, 94]. (2) *GPU sharing* (i.e., trial packing) allows multiple trials to share the GPU using the NVIDIA MPS [13] or MIG [12] technologies to achieve higher utilization [125]. Different from them, Hydro combines scaling, fusion and interleaving for ultimate efficiency.

## 2.2 Hyperparameter Transfer Theory

Recently, the remarkable success of foundation models has ignited a keen interest in exploring the relationship between model size and its optimal hyperparameter. *Scaling Laws* [42, 43, 52] empirically study the power-law functions of batch size and learning rate across varying model sizes. Nevertheless, the authors [52] candidly admit that only limited configurations are tested and the rule-of-thumb formulas break down when dealing with models that exceed one billion parameters.

Beyond heuristic exploration, some novel hyperparameter transfer strategies [49, 117, 121] are proposed by DL theorists. For simplicity, we call them *parametrization*, the rule of how to adjust hyperparameters accordingly when models grow/shrink in both the width and depth. Different from existing HPO systems, Hydro enables automatic hyperparameter transfer based on parametrization. To make the obscure theory more accessible, we present a concise background overview of the underlying theory. [116] systematically builds a coherent theoretical framework for parameterization: the feature learning effect  $\gamma$  of a MLP model is proportional to

$$\gamma \equiv \frac{L}{w^{1-p}}, \quad p \in [0, 1] \quad (1)$$

where  $w, L$  indicates the width and depth of the neural network respectively. For the purpose of simplicity, we assume that the numbers of the hidden-layer neurons are all of similar order,  $w_1, w_2, \dots, w_{L-1} \sim w$ .  $p$  is a metaparameter that interpolates different parametrization strategies into a unified framework, which is determined by inherent strategy. The objective is two-fold: first, to maintain a fixed  $\gamma$  that allows hyperparameters transfer across different model sizes, and second, to strive for a larger  $\gamma$  that facilitates better feature learning. To this end, there are two directions of parametrization:

(1) *Neural Tangent (NT) parametrization* ( $p = 0$ ) [49]. It naturally arose from the study of infinite-wide neural network as Neural Tangent Kernel (NTK) [49, 89], which can keep  $\gamma$  fixed by scaling the depth along with the width as  $L \sim w$ . NTK is a kernel method to explain the evolution of neural networks during training, which is derived by applying the first-order Taylor expansion to linearized models. It belongs to the *lazy training* regime where the weights move very little [121], so that linearization approximately holds around the initial parameters and does not learn features, which is a fatal weakness of the NTK theory in practice. Moreover, NT parametrization does not make sense since the wider model does not always perform better in this context [117], which conflicts with common observations [43, 52].

(2) *Maximal Update (MU) parametrization* ( $p = 1$ ) [121].

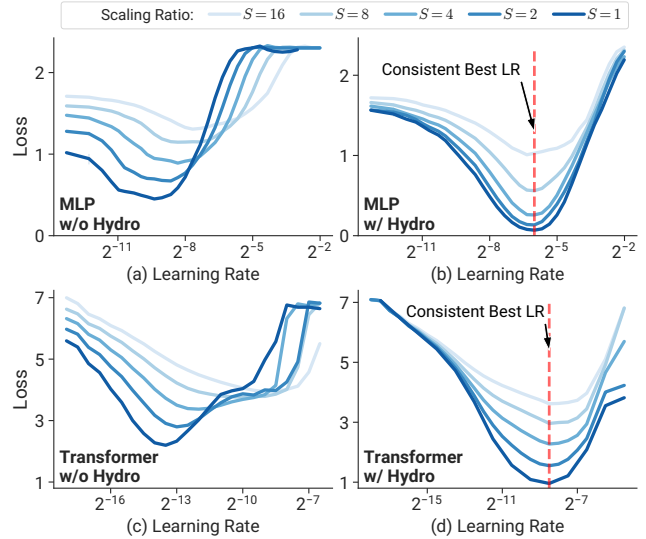


Figure 1: Effect of Hydro parametrization. The training loss against the learning rate on MLP (a, b) and Transformer (c, d) with different widths.  $S$  denotes the model scaling ratio.

It generalizes the mean-field limit of the 1-hidden-layer case [25, 80] and should be the unique parametrization that retains the representation-learning capability (non-rigorously referred to *active training*, in contrast to lazy training of NT parametrization) for a large-scale neural network, which means training does not become trivial or stuck at the initialization in the large width limit. Colloquially, it is designed to solve the issue that the input layer is updated much more slowly than the output layer, and make all hidden activations update with the same speed in terms of width [117].

Hydro adopts the MU parametrization, which will be further elaborated in §4.1 and we refer readers to [98, 117–122] for a comprehensive review of the theory.

## 2.3 Opportunities for Efficient Tuning

**Lightweight surrogate-based tuning.** Current HPO systems search hyperparameters directly on the *target model*, which is intuitive but inefficient. In contrast, Hydro makes it possible to tune a model with a much smaller *surrogate model* via applying a novel hyperparameter transfer technique (aforementioned in §2.2). For a clearer illustration of the surrogate-based tuning effect, we employ Hydro parametrization on two toy models and plot their converged training loss against a range of learning rates as shown in Figure 1. Specifically, the target MLP model contains two hidden layers (width=4096) and we train it with SGD on CIFAR-10. Similarly, the target Transformer model contains two TransformerEncoder layers (width=4096, i.e.,  $d_{model}$ ) and we train it with Adam on WikiText-2. Besides, we generate surrogate models with different scaling (shrinking) ratios  $S$ , and the smaller model is depicted by the lighter blue line. For instance,  $S=2$  represents the model with width=2048. Obviously, the conventional training paradigm (Figure 1 (a, c)) cannot share the best hy-

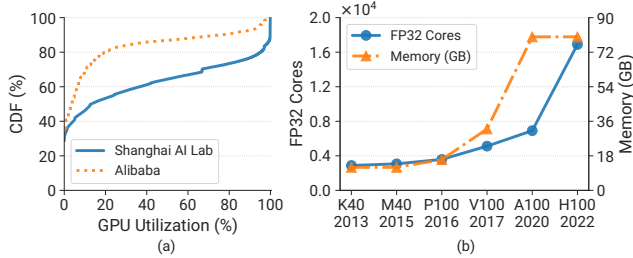


Figure 2: (a) GPU utilization distribution of one partition in our cluster and a GPU production cluster in Alibaba [115]. (b) Exponential growth of NVIDIA datacenter GPU capability. x-axis: GPU model name & release year.

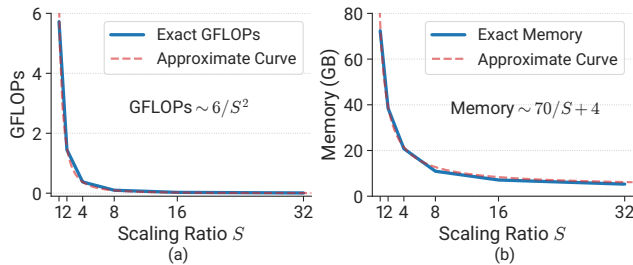


Figure 3: Model scaling effect of WideResNet-50. (a) Model GFLOPs (Giga Floating Point Operations). (b) GPU memory.

perparameter across different sizes of models and there are orders of magnitude optimal learning rate shifts. However, Hydro parametrization (Figure 1 (b, d)) makes surrogate models stay approximately the same optimal learning rate as the target model, which implies the feasibility of surrogate-based tuning. Furthermore, Hydro parametrization can deliver better performance since both tuned MLP and Transformer achieve lower training loss than their counterparts. An intuitive explanation is that the learning rate of the conventional paradigm must tame logits’ surge, but preceding layers do not learn appreciably. We perform a comprehensive evaluation of several models in §6.3 and demonstrate the superiority of Hydro.

**Fusion of numerous repetitive models.** GPUs are commonly underutilized in DL clusters [45, 46, 115, 124, 125]. Figure 2 (a) plots the Cumulative Distribution Function (CDF) of one-week GPU utilization in one partition of our cluster, as well as an Alibaba trace [115] for reference. We find there are only 16% and 35% of GPUs achieving higher than 50% GPU utilization in Alibaba and Shanghai AI Laboratory respectively. This issue will be exacerbated if the Hydro surrogate-based tuning technique is applied. For instance, Figure 3 presents the model scaling effect of training WideResNet-50 on ImageNet, where GFLOPs follows approximately inverse-square ( $c_1/S^2$ ) trend drop and memory footprint follows roughly  $c_1/S + c_2$  trend decrease ( $c_i$  indicates constant). This implies model scaling can significantly reduce the computation overhead, but resources are more prone to be underutilized. To this end, inspired by JAX vmap function [35, 112], Hydro implements an *inter-trial fusion* mechanism to automatically combine multiple models into one. Operators of multiple trials can be

fused owing to the property of HPO tasks: essential is a set of identical models (or with minor mutation). Compared with the conventional GPU sharing mechanism (e.g., MPS), Hydro can achieve higher training throughput, GPU utilization and lower memory footprint (Figure 8).

**Cluster resource awareness.** Although HPO jobs are pervasive in GPU datacenters, cluster schedulers typically regard them as general training workloads without any specific design. On the other hand, HPO systems [9, 72, 84] are cluster resource agnostic. This causes cluster-level inefficiency, such as long job queuing delay and low GPU utilization. However, the unique features of HPO jobs bring opportunities for more efficient tuning. (1) *Trial throughput insensitivity.* Unlike general DL jobs, HPO jobs are more tolerant to throughput slowdown of partial trials. Therefore, we can run more trials by leveraging ephemeral bubble resources of large language model training jobs, which are long-term existing in our datacenter (§5.1). (2) *Diminishing resource requirements.* Multi-fidelity HPO jobs usually explore plenty of trials at the beginning and gradually decrease the search concurrency [32, 71, 82]. At the final stage, only a few trials are exploited. Therefore, we can not only reduce the total resource amount progressively, but also properly leverage the heterogeneous GPU resource (§5.2). With the rapid evolution of GPU computing capability as shown in Figure 2 (b), they become more prone to be underutilized for most small-scale trials [87]. Allocating trials to appropriate GPUs can significantly improve cluster-wide efficiency without hurting a single HPO job makespan.

### 3 Hydro Overview

**Design principles & goals.** For practical adoptions, Hydro follows three design principles: (a) *Automatic and simple.* Manually converting surrogate models is tedious and error-prone. Hence, the whole tuning workflow should be automated and easy to use, which requires minimum user code modification. (b) *Incentive and interference-free.* Although our system focuses on optimizing HPO jobs, it does not sacrifice other workload performance. Instead, it is altruistic and requires fewer resources than conventional systems, which benefits all cluster users. (c) *Modular and extensible.* Each component in Hydro can work independently to support more scenarios (e.g., cloud). Moreover, Hydro can be applied to general HPO tasks, and more tuning algorithms can be easily integrated. In addition, Hydro has two primary objectives: (1) minimizing the makespan of HPO workloads; (2) improving the cluster-wide resource utilization.

**System architecture.** Figure 4 depicts the architecture of the Hydro service. It consists of two key system components: *Hydro Tuner* (blue block) as a user interface to automatically generate surrogate models and optimize tuning trials, and *Hydro Coordinator* (purple block) for improving tuning efficiency and datacenter-level resource utilization. Each component contains several modules for different purposes. Specifically, there are three main modules in Hydro Tuner:



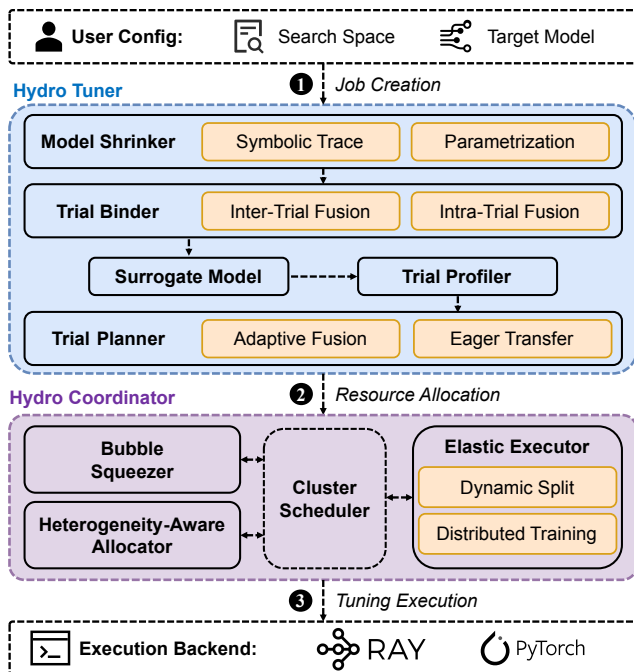


Figure 4: Overview of Hydro architecture and workflow.

- *Model Shrinker*: to obtain surrogate models by automatically tracing, scaling and parametrization.
- *Trial Binder*: to better utilize accelerators by binding multiple trials and fusing internal operators.
- *Trial Planner*: to adaptively determine the tuning strategy based on the profiling information and intermediate results.

Additionally, Hydro Coordinator also includes three modules:

- *Bubble Squeezer*: to extend tuning workload resources by interleaving training with a pipeline-enabled large model.
- *Heterogeneity-Aware Allocator*: to improve resource utilization and cluster-wide performance by allocating proper accelerators on different tuning stages.
- *Elastic Executor*: to dynamically execute trials by splitting fused trials and enabling distributed training.

**API Design.** Hydro enables high-efficient surrogate-based hyperparameter tuning with a few lines in the developer’s code, as shown in Figure 5. It follows the Ray Tune [72] API to define the search space and invoke the `fit()` function. To support Hydro functions, developers only require to wrap their model, dataloader and optimizer with the `prepare_xxx()` API (lines 6~8). Hydro traces the whole function to control the trial execution, convert surrogate model, enable model fusion and elastic training.

**Tuning Workflow.** The system workflow of Hydro is presented by black arrows in Figure 4. Specifically, when a developer wants to tune a model, she only needs to define the search space and invoke the Hydro APIs (①). After job creation, Hydro Tuner automatically generates and optimizes surrogate models by scaling and fusion. Furthermore, it adopts *Trial Planner* to efficiently orchestrate the tuning process. Then Hydro Coordinator is responsible for contacting the cluster

```

1 import ray, hydro
2 import hydro.train as ht
3
4 def train_func(config):
5     # Wrap model, dataloader and optimizer
6     model = ht.prepare_model(model)
7     data_loader = ht.prepare_data_loader(data_loader)
8     optimizer = ht.prepare_optimizer(SGD, lr=config["lr"])
9     for _ in range(1): # User defined training loop
10        train_epoch(...)
11        result = validate_epoch(...)
12        ray.session.report(result)
13
14 search_space = {"lr": ray.qloguniform(1e-4, 1, 1e-4)}
15 trainer = hydro.Trainer(train_func)
16 tuner = hydro.Tuner(trainer, search_space, scaling_num=8)
17 results = tuner.fit()

```

Figure 5: A code example of how to use Hydro APIs to define the search space and perform hyperparameter tuning.

scheduler to dynamically allocate resources and execute trials (②). It supports two novel mechanisms, which leverage ephemeral bubbles and heterogeneous resources to further improve datacenter efficiency. Finally, the tuning job is successfully scheduled and starts running, where Ray [84] and PyTorch [91] serve as the execution backend (③). More details are introduced in the following sections (§4 & §5).

## 4 Hydro Tuner

Hydro Tuner is a core component of the Hydro service for job-level optimization. It consists of three modules: Model Shrinker, Trial Binder and Trial Planner.

### 4.1 Model Shrinker

Model Shrinker aims to obtain surrogate models by automatically tracing, scaling and parametrizing the target model. The upper part of Figure 6 depicts its workflow. It first traces the target model and edits each layer’s configuration to build a scaled model (①). To enable hyperparameter transfer, it then automatically parametrizes the scaled model by reinitializing the weight and adjusting the learning rate of each layer accordingly (②). Below we first summarize the MU parametrization theory that Hydro parametrization relies on, and then introduce how Hydro brings it into practice.

**Maximal Update parametrization.** As introduced in §2.2, Hydro employs the MU parametrization theory [117, 121] to search hyperparameters on a small surrogate model and transfer them to the large target model. The theory is built atop Tensor Programs [117–122], a unified theoretical framework that formulates the computation of common neural networks components as Gaussian Processes (GPs), including multi-layer perceptrons (MLPs), recurrent neural networks (RNNs) (e.g., Long-Short Term Memory (LSTM) [21]), skip connections [41], convolutions [62] or graph convolutions [55], pooling [62], batch normalization [48], layer normalization [20],



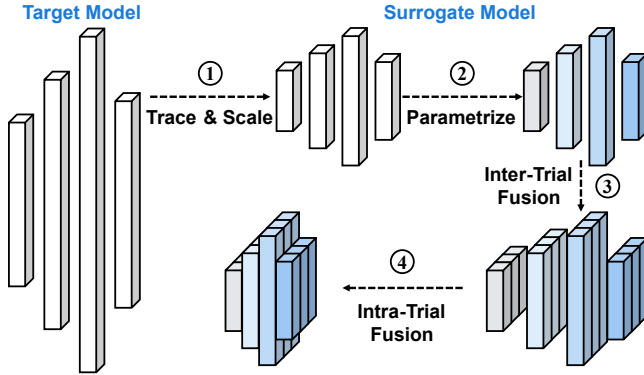


Figure 6: Illustration of Model Shrinker (①, ②) and Trial Binder (③, ④). The length of each bar represents layer width.

and attention [108]. As a result, many practical models like ResNet [41] and Transformer [108] can be expressed as GPs and apply MU parametrization, since they inherently consist of these basic components.

► **Theory assumption.** In contrast to prior works such as NTK [49] that necessitate unnatural conditions, MU parametrization only requires standard Gaussian initialization for the model, which is easily achievable in practice. In terms of data, MU parametrization requires i.i.d. samples, which is typically present in the same dataset. However, this requirement also limits its ability to support fine-tuning (§7).

► **Key insight and mechanism.** The main idea of MU parametrization is: every activation vector has roughly i.i.d. coordinates at **any time** during training neural networks in the infinite-width limit. It aims to overcome the imbalanced per-layer learning speed issue in practice. To this end, MU parametrization performs layer-wise fine-grained adjustment, including per-layer initialization variance, learning rate and other optimizer-related hyperparameters (e.g., SGD momentum, Adam beta). Specifically, since the output layer is updated much faster than the input layer, MU parametrization suppresses the learning rate and initialization variance of output weights by  $w$  (width) times. In addition, for SGD-like optimizers (linear tensor update), the learning rate of input weights and all biases is multiplied by  $w$ . For Adam-like optimizers (non-linear tensor update, normalizes the gradient coordinate-wise), the learning rate of hidden weights is divided by  $w$ . Hence, MU parametrization ensures consistent magnitude updates for each layer during training regardless of its width so that hyperparameters can be transferred across models with different widths at any time (i.e., same converge speed across scaled models).

To summarize, in the large width limit, MU parametrization reveals that hyperparameters yielding lower training losses for narrower models also result in better performance for wider models through a specific transfer mechanism. Hydro leverages this effect to obtain better test accuracy efficiently via surrogate-based tuning, albeit without a rigorous theoretical guarantee for every model.

► **Instructive example.** To provide a clearer explanation of why parametrization is necessary and how it operates, we recapitulate the key insights of [121] with an instructive example [117]. Consider a 1-hidden-layer linear model  $f(x) = V^T U x$  with scalar inputs and outputs, as well as  $w$ -width layer weights  $V, U \in \mathbb{R}^{w \times 1}$ . In common practice (e.g., Xavier initialization [37]), we initialize them with  $V \sim \mathcal{N}(0, 1/w)$  and  $U \sim \mathcal{N}(0, 1)$ , which ensures  $f(x) = \Theta(|x|)$  at initialization ( $\Theta(\cdot)$  indicates asymptotically tight bound). After one step of SGD with learning rate 1, the new weights are  $V' \leftarrow V + \theta U$  and  $U' \leftarrow U + \theta V$ , where  $\theta$  is some scalar of size  $\Theta(1)$  depending on the inputs, labels, and loss function. Then

$$\begin{aligned} f(x) &= V'^T U' x \\ &= \left( V^T U + \theta U^T U + \theta V^T V + \theta^2 U^T V \right) x \end{aligned} \quad (2)$$

which will blow up with width  $w$  in the infinite limit because  $U^T U = \Theta(w)$  by Law of Large Numbers. In other word, it only allows  $O(1/w)$  learning rate so as to avoid float overflow, and yield kernel limits (§2.2). Consequently, it fails to perform feature learning (learning rate  $\rightarrow 0$ ) to update weights after random initialization.

However, by applying maximal update parametrization, we have  $V \sim \mathcal{N}(0, 1/w^2)$ ,  $U \sim \mathcal{N}(0, 1)$ , learning rates  $\eta_V = 1/w$  and  $\eta_U = w$ . After one step of SGD, now we have

$$f(x) = \left( V^T U + \theta w^{-1} U^T U + \theta w V^T V + \theta^2 U^T V \right) x \quad (3)$$

and one can verify this is  $\Theta(1)$  and remains bounded. In contrast to common practice, MU parametrization has  $\Theta(1)$  learning rate and admits feature learning *maximally*, which allows every parameter to be updated maximally (in terms of scaling with width) without leading to float overflow.

► **Heuristic adaptation.** While Tensor Programs support more versatile model components (e.g., convolution), obtaining a closed-form solution for arbitrary models is infeasible. The efficacy of the MU parametrization has been rigorously demonstrated on a 2-hidden-layer MLP trained with SGD for multiple steps, and the proof can be readily extended to deeper MLPs [121]. For more general models in practice, some heuristic tricks are adopted to enhance their hyperparameter transferability. For example, Transformer [108] models require two additional operations in the self-attention: (1) scaling the attention logit by  $1/d_k$  rather than  $1/\sqrt{d_k}$ , where  $d_k$  is the attention head size; (2) zero initialization on query layer  $q$ . We also empirically find that using a larger sequence length provides a better transfer effect for Transformer models. For models with some special components or architecture (e.g., MoE [101]), hyperparameters may not well transfer with MU parametrization alone. Hence, additional analysis and tailored adjustments may be required.

**Hydro parametrization.** It is arduous and error-prone to implement MU parametrization manually to generate a surrogate model. Developers are required to not only thoroughly understand the MU parametrization theory, but also manually

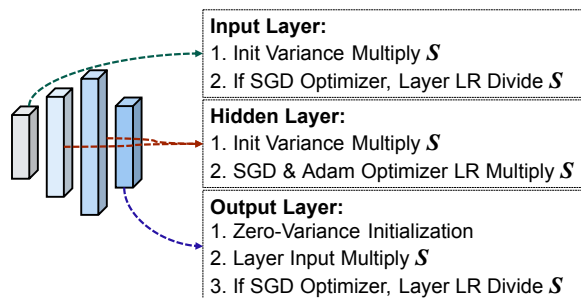


Figure 7: Hydro parametrization implementation. Illustration on a simple 4-layer model with SGD or Adam-like optimizer.

adjust the model width, initialization function and learning rate layer by layer. Any incorrect adjustment may directly incur hyperparameter transfer failures. To this end, we implement Hydro parametrization, an automated and simplified parametrization strategy based on MU parametrization. We demonstrate the excellent effect of Hydro parametrization with visualized results in Figures 1 and 10.

For a clearer illustration, we present the Hydro parametrization process in Figure 7, which applies different strategies to the input, hidden and output layers. Developers only need to specify their desired *scaling ratio*  $S$  ( $S = 8$  by default) and then Hydro will parametrize the model accordingly. Concretely, at the model initialization stage, we apply zero-variance initialization to the output layer instead of  $1/w^2$ , which will not be detrimental to performance and can remove this mismatch issue between the surrogate model and target model in the initial Gaussian process [117]. Moreover, we apply zero initialization to all biases, and weights as well as learning rate scaling strategies are annotated in the figure, which is invoked by the `prepare_optimizer` API to build a `hydro_optimizer`. Besides, we insert a Multiply layer in front of the output layer to scale its input by  $S$ .

**Applicable Scope:** Hydro parametrization works well for most ubiquitous hyperparameters that control model initialization and training, including learning rate, batchsize, `lr_scheduler`, momentum, etc. However, it has limited support on regularization-related hyperparameters, such as weight decay and dropout, because they naturally depend on both the model size and data size. Although parametrization cannot be applied to all hyperparameters, it is sufficient to achieve qualified performance in most cases. After most hyperparameters are tuned with the surrogate model, developers can further tune the regularization hyperparameters within a much smaller search space on the target model if needed. Moreover, we provide a comprehensive summary of additional limitations associated with Hydro parametrization in Section 7.

**Trace and scale.** Before performing the above parametrization, we need to first trace the target model and build a scaled model. Since there are various model definition styles in the PyTorch ecosystem, it is necessary to obtain a uniform and equivalent modality from disparate community model codes. We implement HydroTracer based on `torch.fx` [97], which

allows developers to trace and edit the model. Specifically, we replace `call_function` nodes (e.g., `torch.nn.functional`) with the corresponding `call_module` nodes (e.g., `torch.nn`) for subsequent layer scaling and fusion (§4.2). We apply different scaling rules to the input, output and hidden layers. For instance, we parse `nn.Linear` kwargs and modify both the `in_features` and `out_features` values by dividing  $S$  for hidden layers. In addition, we only scale the `out_features` of input and `in_features` value of output layers. To handle the data-dependent control-flow, we use proxy nodes along with developer-provided concrete values to determine the execution flow [61]. According to our evaluation of notable models, including torchvision [18] (e.g., ResNet [41], MobileNet [44], VGG [103]) and HuggingFace Transformers [113] (e.g., BERT [30], GPT [95], Swin [76]), developers can trace and scale these models with Hydro without modifying the code.

**Correctness check.** While Hydro has achieved automatic parameterization, there are still potential failures due to certain special model components that require heuristic adaptation as previously mentioned, as well as other corner cases that have not been considered. To this end, we further implement a safeguard mechanism to check the correctness of the parametrization and notify users whether they should use Hydro to prevent misleading hyperparameters and resource wastage. Firstly, Hydro performs a simple per-layer width check when scaling to avoid too narrow layers (e.g., only 1 neuron width for a Linear layer). Additionally, taking inspiration from gradient checking as a simple method for verifying the correctness of an autograd implementation, Hydro has a quick parameterization profiling stage that checks whether the average size (L1 value) of each activation vector is bounded to avoid possible parameterization failure based on [117]. It only lasts for very few steps at the beginning of the HPO job.

## 4.2 Trial Binder

Although Model Shrinker dramatically reduces the computation of each trial (Figure 3), it inevitably incurs the resource underutilization issue, which deteriorates small- or mid-size target models (e.g., deployed on edge devices). To address this problem, Trial Binder further optimizes surrogate models by binding multiple trials and fuses internal operators to better utilize accelerators. We illustrate its mechanism in the bottom part of Figure 6. It merges a set of fusible trials into a *HydroTrial* with grouped operators and optimizer (③). To further accelerate training, we automatically just-in-time (JIT) compile the fused (*inter-*) surrogate model to obtain fast and flexible fusion (*intra-*) kernels (④). Note that the last model with closer layer distance represents the reduced memory-bounded operations through intra-trial fusion.

**Inter-trial fusion.** There are plenty of trials with the same or similar model structure in a HPO job. Inspired by JAX `vmap` [35, 112], which returns a batched version of the target function by vectorizing each input along the axis specified, we can batch multiple trials into a single one by fusing their opera-

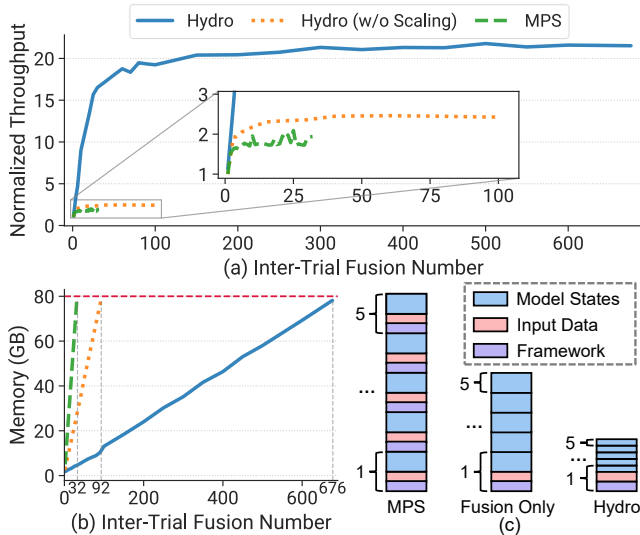


Figure 8: Inter-trial fusion effect on ResNet-18. (a) Accumulated throughput of fused surrogate model w.r.t the target model. (b) GPU memory footprint of different fusion counts. Red horizontal line denotes the A100 memory bound. (c) Schematic diagram of memory occupation detail of 5 models GPU sharing with MPS, Hydro and Fusion (w/o Scaling).

tors. Hydro implements an *inter-trial fusion* mechanism to automatically bind surrogate models. Specifically, Trial Binder traverses the traced surrogate model and replaces the `torch.nn` operators with grouped `hydro.nn` operators according to the predefined fusion rule and fusion count  $F$  determined by Trial Planner. `hydro.nn` provides mathematically equivalent implementations of batched original PyTorch operators based on HFTA [110]. For instance, `hydro.nn.Linear` is implemented atop `torch.baddbmm` (i.e., batch matrix-matrix product and add), which adds an additional dimension batch (i.e.,  $F$ ) compared with `torch.nn.Linear (addmm)`. Besides, for each `hydro.nn` operator, we reimplement the initialization function to support independent model-wise Hydro parametrization and realize the defusion mechanism to extract a specific sub-model. Additionally, `hydro_optimizer` and `hydro_lr_scheduler` are designed to support both the model fusion and parametrization simultaneously. These are performed automatically, and developers typically do not need to understand the rationale and modify codes.

Figure 8 plots the extraordinary effect of integrating model scaling with inter-trial fusion on ResNet-18 ( $S = 8$ ), tested on CIFAR-10 with batchsize=256. It is evident that Hydro is capable of concurrently training impressive 676 models on a single A100 GPU. Compared with the conventional GPU sharing mechanism MPS [13] (MIG [12] has similar performance), Hydro achieves over  $10\times$  training throughput improvement and over  $20\times$  GPU memory conservation. If we directly apply inter-trial fusion to the target model (without scaling), the throughput improvement is relatively much limited. Furthermore, we provide an intuitive interpretation

of how memory footprint reduction occurs in Figure 8 (c). The model states (blue blocks) encompass all aspects associated with model training such as model weights, gradients, activations, and optimizer states [96]. MPS has repetitive memory overheads incurred by CUDA context of DL framework (purple blocks) and independent data loading (pink blocks). In contrast, Hydro avoid such redundancy and further reduce model-related memory footprint. Note that here we only compare with vanilla training paradigm without considering more advanced memory optimization techniques like Salus [124]. Moreover, beyond the better GPU utilization and higher throughput, inter-trial fusion also alleviates the I/O pressure owing to the accompanied data-loading fusion.

**Lazy intra-trial fusion.** Hydro supports automatic model fusion to further accelerate training based on the nvFuser [10] compiler backend. Although plenty of previous works [51, 107, 129] demonstrate that operator fusion can improve training throughput via better memory locality, it does not always bring benefits to HPO workloads due to its high compiling overhead. For instance, nvFuser [10] takes approximately 2-epoch time to compile a ResNet-18 model to deliver around 10% speedup per epoch, which means a trial needs to run at least 20 epochs to avoid slowdown. However, most trials will end up in a few epochs for multi-fidelity tuning algorithms. To this end, Hydro apathetically adopts the intra-trial fusion. For simplicity, Hydro currently only applies to trials with *deterministic* training steps, such as all HydroTrials when applying single-fidelity tuning algorithms and the trial that trains the target model with transferred hyperparameters.

### 4.3 Trial Planner

Trial Planner is the key module that interacts with the tuning algorithm and trial executor. We introduce two mechanisms that improve the surrogate-based tuning efficiency.

**Adaptive fusion.** The trial count and resource amount vary significantly across different HPO jobs. Hence, the fusion count  $F$  of each HydroTrial should be adaptively determined to achieve the desired performance. Hydro contains the following steps to fuse trials and assign GPUs: (1) Trial Planner invokes the tuning algorithm to generate a set of hyperparameter configurations (trials). (2) Since inter-trial fusion requires trials with the same operator shapes, we split them into different *trial groups* according to their batchsizes. (3) Based on the linear growth of GPU memory shown in Figure 8 (b), we can profile the trials with  $F = 1$  and  $F = 2$  for each *trial group* and estimate the upper bound of the fusion count  $F_{max}$ . (4) Hydro assigns all available GPUs to each *trial group* according to group’s weight, which equals to  $B \times N$  (denoted as the product of batchsize and trial count of the group). (5) Each *trial group* evenly distributes trials based on the group GPU amount and  $F_{max}$ , and Hydro fuses them as a HydroTrial on each GPU. In this way, Hydro can leverage as many GPUs as possible and achieve the optimal global throughput.

**Eager transfer.** As the HPO job progresses, more and more



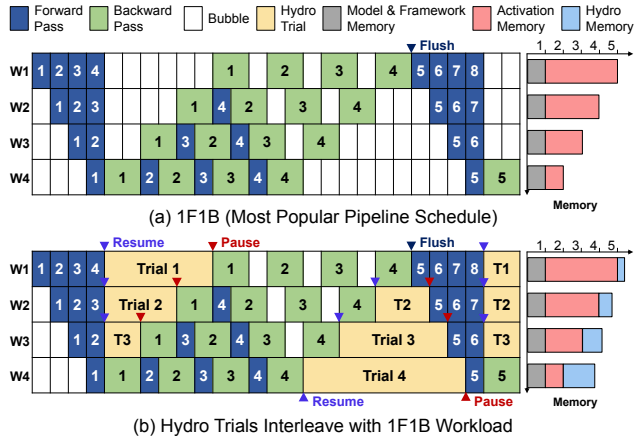


Figure 9: Illustration of (a) 1F1B Pipeline and (b) Hydro Bubble Squeezer, with four pipeline stages and four micro-batches. Note the right-side memory diagrams can only reflect the relative relation of the *same* color blocks across workers.

trials terminate and the degree of the parallelism gradually decreases, resulting in underutilized or idle resources. On the other hand, the best hyperparameter configuration sometimes appears in the early stage. Therefore, instead of training the target model after all the surrogate-based tuning trials are done, we can eagerly transfer the intermediate best hyperparameters and leverage vacated resources to validate the configuration on the target model. Hydro records all evaluated hyperparameters and schedules a *TargetTrial* for the target model training when 50% (customizable) of the surrogate-based tuning trials are done and there exist idle resources. If a better hyperparameter is searched, Hydro terminates the on-going *TargetTrial* or starts a new *TargetTrial* depending on the resource utilization. This mechanism efficiently shortens the job makespan and improves the resource utilization.

## 5 Hydro Coordinator

Hydro Coordinator focuses on cluster-level optimization. It consists of three modules: Bubble Squeezer, Heterogeneity-Aware Allocator and Elastic Executor. It is important to highlight that the first two modules are tailored for specific cluster scenarios. Specifically, Bubble Squeezer can only be activated when a pipeline-enabled foundation model pretraining job is running within the cluster. The Heterogeneity-Aware Allocator is meticulously designed to better leverage multiple generations of GPUs coexisting in the cluster.

### 5.1 Bubble Squeezer

In addition to HPO jobs, there are many kinds of workloads that coexist in the GPU datacenter, such as inference, debugging and large-scale distributed training jobs [45, 50, 111]. With the rapid popularity of foundation models (e.g., GPT-3 [24]) in recent years, some large model pretraining workloads exist in our datacenter in the long term. As complained by many users, the majority of machines are occupied by large model training jobs that usually last for days to weeks,

which incurs the starvation of other jobs. Additionally, the pipeline parallelism [85, 88] is usually adopted to support a larger model by splitting it into several stages and placing them across multiple workers. However, bubbles inherently exist in the synchronous pipeline parallelism [106], such as the commonly used 1F1B [34, 86] strategy. Besides, the imbalance peak memory issue (Figure 9) between different pipeline stages further exacerbates the resource inefficiency [65].

Hydro designs Bubble Squeezer, which leverages bubbles to greatly extend the tuning job resources in an interleaving execution way, almost without hurting the training throughout of large models. HydroTrials are perfectly suitable for the bubble interleaving execution due to the following unique features: (1) *Throughput insensitivity*. Unlike general DL training jobs, tuning jobs are more tolerant of the slowdown of partial trials. This inspires us to squeeze the spare resources of the bubbles and execute trials in a pause-and-resume way. (2) *Deterministic resource pattern*. General small-scale workloads (e.g., debugging) have unknown and unpredictable resource requirements. However, Hydro profiles and records the resource consumption of HydroTrials, mitigating the potential out-of-memory (OOM) issues if they are colocated with large models. (3) *Elastic trial size*. Based on Model Shrinker, the scaled model has a much smaller memory footprint (Figure 3) than the original model, which means we typically do not need to swap out its GPU memory during collocation. Besides, we can dynamically adjust the trial fusion count according to the remaining GPU memory with Trial Binder.

To clearly illustrate how Bubble Squeezer works, we first introduce the 1F1B pipeline parallelism in Figure 9 (a). It transfers intermediate activations of the partial model at the forward and backward passes between different workers using point-to-point communication [130], thus each worker cannot continuously utilize the GPU. For Worker 1, after the forward pass of the last micro-batch (blue block 4), it has to wait for the backward pass of the first micro-batch (green block 1), leaving GPU idle for a long time. Other workers also present similar bubble patterns but occupy less GPU memory since fewer activations of micro-batch needed to store.

In Figure 9 (b), Hydro interleaves four HydroTrials of different sizes with the large model training workload. Each trial executes in a pause-and-resume paradigm to squeeze the bubbles. Since Hydro Tuner has traced and canonicated each layer with `hydro.nn`, we further register hooks on each module of the trial to support on-demand pause and resumption in the forward and backward passes of each layer. When a large model training job exists, Hydro coordinates with datacenter scheduler to acquire more GPUs from this model and tags them as *ephemeral* resources. For the large model, we also implement a corresponding hook inside its training framework (i.e., DeepSpeed [96]) to report its training progress and resource consumption. Each worker executes its corresponding pipeline under DeepSpeed’s pipeline parallelism. Therefore, we implement a fine-grained synchronization mechanism to



guarantee that `HydroTrials` only could be executed within the bubbles, by intercepting the status of the CUDA streams of the NCCL kernels. Hydro can further adjust the fusion count to adaptively fit in the remaining memory and improve GPU utilization. At the beginning and end of the bubble of large model training, we control the resumption and pause of trial model training by Linux signals. The fine-grained suspend-resume control eliminates the performance interference caused by CUDA kernels running simultaneously.

In general, the effectiveness of Bubble Squeezer varies depending on multiple factors, and we present the scenarios where it works best. Regarding the HPO job aspect, Hydro is more effective when using (1) *multi-fidelity tuning algorithms* because they allow most trials to be terminated in a few epochs using the ephemeral resources and execute immediate top trials on exclusive resources to avoid possible blocking caused by interleaving slowdown. In addition, (2) *models with fewer layers* are preferred as they are prone to complete the entire iteration within the bubble time and require relatively less memory to support a higher fusion number. As for pipelined large model aspect, Hydro can achieve better performance when the pretraining job has (3) *more pipeline stages across more servers*, which implies a higher bubble ratio and more ephemeral resources. A large model pretraining job typically can support multiple different HPO jobs interleaving simultaneously and accelerate dozens, even hundreds of HPO jobs (depending on its resources and duration scale) during its pretraining process. In addition, there may be cases where some scaled models are still too large to be allocated on any GPU of the pretraining model. Due to the high memory swap overhead in our scenario, Hydro does not support offloading techniques like Bamboo [106]. As a result, Bubble Squeezer is unable to support such models.

## 5.2 Heterogeneity-Aware Allocator

HPO workloads generally have diminishing resource requirements [71]. They usually explore plenty of trials at the beginning and gradually decrease the search concurrency. At the final stage, only a few trials are exploited. Hence, tuning with fixed GPU resources can lead to underutilization. Existing HPO systems [32, 82] support autoscaling to dynamically adjust the tuning resources. However, they do not consider the GPU heterogeneity in the datacenter.

Inspired by Gavel [87], a novel heterogeneity-aware cluster scheduler for general DL jobs, we design a resource allocator to allocate appropriate GPUs to trials, which can improve the cluster-wide efficiency without sacrificing the job makespan. Hydro supports both resource autoscaling and heterogeneity-aware allocation. Specifically, if there is any node or GPU idle for over 1 minute (customizable), Hydro will interact with the cluster scheduler to release the resource. Other affiliated resources like CPU will also be released as a bundle. Additionally, Hydro creates *TargetTrial* with the eager transfer mechanism and makes the target model training process well

hidden inside the tuning time. Since the *TargetTrial* typically trains alone without fusion, it may not be able to fully utilize the GPU resources. So Hydro will place it on an GPU of old version (e.g., V100) if its SM Activity rate (measured by NVIDIA DCGM [11]) is lower than 50% (customizable). Similar action will be applied to surrogate models if their allocated resources are underutilized and there exist other HPO jobs pending in our service queue.

## 5.3 Elastic Executor

Elastic Executor is designed to improve the job efficiency by leveraging all assigned GPU resources. It supports two elastic mechanisms: (1) dynamic split and (2) automated distributed training. Specifically, when an idle GPU emerges, the *fused* `HydroTrial` will not directly increase its GPU count by conventional distributed training. Instead, Hydro will evenly split this `HydroTrial` into multiple `HydroTrials` and exclusively place them on the idle GPUs to reduce the communication overhead. Furthermore, since the memory footprint of some large models is high even though scaled, Hydro supports two types of elastic strategies for *unfused* surrogate models: (a) *Evenly distribute*: allocating idle GPU resources to all unfused surrogate models evenly. (b) *Performance-aware* (default): allocating idle GPU resources to the top performing trial. For the target model, Hydro automatically increases the number of workers to enable distributed training.

## 6 Evaluation

Hydro is implemented on top of Ray [72, 84] with about 12K LoC. For Hydro Tuner, Model Shrinker relies on torch.fx [97] and mup [117], while Trial Binder is built with HFTA [110] and nvFuser [10]. As for Hydro Coordinator, we modify DeepSpeed [96] to further support Bubble Squeezer and validate the interleaving execution as a prototype. And the Elastic Executor based on Ray Train as well as PyTorch FSDP [17].

We evaluate Hydro Tuner and Hydro Coordinator independently for a fair comparison. Our experiment search space does not include weight decay because Hydro is unable to transfer regularization hyperparameters, but it is sufficient to achieve qualified performance without tuning it.

### 6.1 Experiment Setup

**Testbed.** We conduct our experiments on a GPU datacenter of Shanghai AI Laboratory. Each node has 8 NVIDIA A100 80GB GPUs, 2 AMD EPYC 7742 CPUs (128 cores) [2] and 1TB memory. GPUs are interconnected to each other by NVLink and NVSwitch [14], and inter-node communication is achieved via NVIDIA Mellanox 200Gbps HDR InfiniBand [7]. All the experiments are conducted on A100 GPUs, unless explicitly stated in §6.5.

**Workloads and search spaces.** We evaluate Hydro tuning performance over six popular CV/NLP models, as listed in Table 2. Specifically, *GPT-3 XL* is a large language model architecture belonging to GPT-3 family. It contains 1.3B parameters and we use an open source implementation by GPT-

Task	Search Space	Model	Dataset	Optimizer	# of GPU	# of Trial	Avg. Time Reduction	Avg. Quality Difference	Size
Language Modeling	1r: $U_{Q\log}(10^{-5}, 10^{-1}, 10^{-5})$	GPT-3 XL [24]	OpenWebText [38]	AdamW	128	100	78.5 ×	-0.48 ppl	XL*
	gamma: $U_Q(0.01, 0.9, 0.01)$	Transformer [108]	WikiText-103 [81]	Adam	8	200	8.7 ×	-0.15 ppl	M
Image Classification	1r: $U_{Q\log}(10^{-4}, 1.0, 10^{-4})$	WideResNet-50 [126]	ImageNet [29]	SGD	32	200	20.3 ×	+1.18% acc	XL*
	momentum: $U_Q(0.5, 0.999, 10^{-3})$	MobileNetV3 Large [44]	Flowers102 [90]	Adam	16	500	12.3 ×	+0.05% acc	L
	batchsize: [128, 256, 512]	VGG-11 [103]	CIFAR-100 [57]	SGD	8	500	10.8 ×	+0.09% acc	M
	gamma: $U_Q(0.01, 0.9, 0.01)$	ResNet-18 [41]	CIFAR-10 [57]	SGD	8	1000	16.2 ×	+0.02% acc	M

Table 2: Summary of (1) workloads used in the evaluation and (2) single-fidelity tuning improvements over Ray. *Model Quality*: ppl indicates perplexity (the lower the better) and acc denotes accuracy (the higher the better). \* For XL tasks, we estimate the time cost of Ray based on simulation and use the official hyperparameter setting as the model quality baseline.

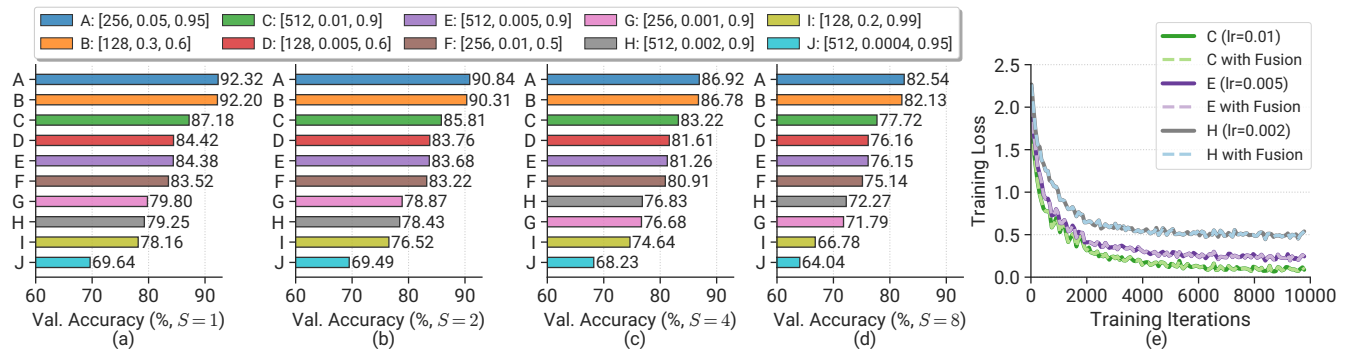


Figure 10: Hydro Tuner mechanisms validation. (a)~(d) *Scaling validation*: randomly select 10 hyperparameter sets ([batchsize, 1r, momentum]) to visualize the transfer effect of multi-dimensional hyperparameters across different scaling ratios  $S = 1, 2, 4, 8$  on model ResNet-18. (e) *Fusion validation*: loss curves of the standard model (solid line) and inter-trial fused model (dash line).

Neo [5, 23]. We further enable mixed precision training for *WideResNet-50* and two language modeling tasks. For the dataset, we crop Flowers102 into  $224 \times 224$  images, whose input size is the same as ImageNet. And we swap its train and test dataset split to get a larger training dataset to make it similar to more general jobs. Moreover, we denote single-node tasks as M-size, and distributed tuning tasks as L/XL-size.

We adopt three kinds of optimizers for above models, including SGD [99], Adam [54], and AdamW [77]. We use StepLR to decay the learning rate (1r) of each parameter group by gamma at every fixed step for all tasks. Additionally, we design two groups of search spaces for CV and NLP tasks respectively (Table 2), where  $U_Q(lower, upper, q)$  represents uniformly sampling a quantized (increment of  $q$ ) float value between *lower* and *upper*. Similarly,  $U_{Q\log}$  uniformly samples in different orders of magnitude. Note that the search space of *MobileNetV3 Large* excludes momentum due to the incompatibility of Adam.

**Tuning algorithms.** Hydro supports multiple popular single-fidelity and multi-fidelity tuning algorithms, such as Random [22], HyperBand [64], ASHA [63]. Since our work focuses on system aspect optimization instead of tuning algorithms, we select two representative tuning algorithms in our evaluation: (1) *Random* (single-fidelity): fully evaluates each randomly generated trial; (2) *ASHA* (multi-fidelity): eliminates

unpromising trials via asynchronous successive halving strategy. They are common hyperparameter tuning paradigms in practice. Besides, their asynchronous and prior-independent nature makes them more suitable for large-scale distributed tuning with numerous trials [71].

**Baselines.** We consider the following two systems as baseline: (1) *Ray* [72, 84]: performs HPO with the vanilla Ray Tune library; (2) *Ray+ES*: applies two advanced techniques in Ray Tune (*Elastic training* and *GPU Sharing*). Our implementation of *Ray+ES* refers to HyperSched [71] and Fluid [125]. Specifically, we place multiple trials on the same GPU using NVIDIA MPS [13] and allocate more GPU resources to the top performing trials if idle GPUs are available. We do not employ A100 MIG [12] sharing due to its similar performance with MPS but less flexibility [110]. Additionally, since existing popular HPO systems (Table 1) mainly differ in the application scenario and API design, and their system performance on the same tuning algorithm is similar, the Ray-based systems are sufficient for representing SOTA.

## 6.2 Surrogate-based Tuning Validation

Before performing end-to-end evaluations, we first give an intuitive experiment to validate the effect of surrogate-based tuning, which is the foundation of Hydro. As shown in Figure 10 (a)~(d), we randomly choose 10 hyperparameter configurations (denoted as A~J) on the ResNet-18 model and build

Model	# of GPU	# of Trial	Avg. Time Improvement	Avg. Quality Difference
GPT-3 XL	64	100	33.4 ×	-0.43 ppl
Transformer	4	200	5.8 ×	-0.09 ppl
WideResNet-50	16	200	9.7 ×	+0.87% acc
MobileNetV3 Large	8	500	8.0 ×	+0.08% acc
VGG-11	4	500	9.4 ×	+0.19% acc
ResNet-18	4	1000	14.5 ×	+0.05% acc

Table 3: Summary of multi-fidelity tuning improvements.

Deadline (s)	# of GPU	Model	Avg. Accuracy		
			Ray	Ray+ES	Hydro
900	4	VGG-11	65.42%	66.39%	<b>68.68%</b>
		ResNet-18	89.66%	90.71%	<b>91.32%</b>

Table 4: Summary of tuning performance with a deadline.

surrogate models with Hydro using different scaling ratios  $S = 2, 4, 8$ , where  $S = 1$  represents the target model. We train each model for 100 epochs on the CIFAR-10 dataset with a fixed seed=1. Since the HPO job is essentially a ranking problem of hyperparameter configurations, we mainly care about whether the order is maintained especially for the top configurations, namely hyperparameter transfer effect. From the result, it is obvious that the performance ranking of hyperparameters transfers well across different scaling ratios. Admittedly, configurations G and H are swapped when  $S \geq 4$ , but it has no influence on the final tuning result since they perform poorly and top configurations keep a consistent ranking. Besides, the wider model always outperforms the narrower one under the same hyperparameters, which is inline with MU parametrization theory and demonstrates that surrogate model can effectively transfer multi-dimensional hyperparameters.

Additionally, we also validate the inter-trial fusion effect, which is another key mechanism of Hydro. Figure 10 (e) shows the training loss curves of trials C, E, H and their fused versions. We select these three trials because their batch size and momentum are consistent and only differ in lr. As we can see, the convergence curves of the fused model well overlap with the original standalone training curves, which demonstrates that inter-trial fusion is a mathematically equivalent transformation and does not affect the model convergence.

### 6.3 End-to-End Performance of Hydro Tuner

To cover most hyperparameter tuning scenarios in practice, we conduct end-to-end experiments across 6 workloads with different settings and 3 common tuning paradigms (case I~III). Note that Hydro Tuner adopts a fixed resource size (without enabling Hydro Coordinator) for fair comparisons.

**Case I: single-fidelity tuning.** When a user seeks for extremely excellent model performance with ample resources, single-fidelity tuning is applied to avoid missing the best hyperparameter configuration. Table 2 summarizes the Hydro

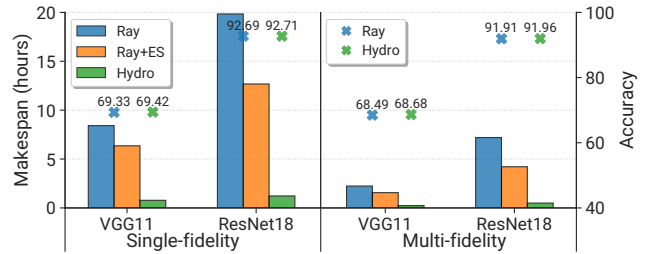


Figure 11: Summary of the end-to-end results. Bars indicate tuning makespan and points represent final model accuracy.

improvement on single-fidelity tuning over different sizes of workloads, where we apply  $S = 16$  for XL models and  $S = 8$  (default value) for other models. Since HPO jobs require completely training massive trials, we perform each experiment twice and report their average results on time reduction and tuned model quality over Ray. Besides, we obtain Ray tuning time of XL experiments based on simulation due to their unacceptable tuning cost, and adopt the official hyperparameter configurations [16, 24] to train the model as quality baselines. The target model training time is included in Hydro.

From the table, we can see that Hydro substantially outperforms Ray by 8.7~78.5× in time reduction, while obtaining better final model quality. The time reduction mainly derives from two aspects: (1) *Less resource demand of trials*. For instance, the scaled GPT-3 XL trials do not require distributed training. For smaller models, Hydro further applies inter-trial fusion to improve trial concurrency and resource utilization. (2) *Smaller model trains faster*. Each trial has fewer FLOPs (Figure 3) to compute, which is more obvious on larger models. Additionally, we also observe that the effect of Hydro is more evident for larger models, with more intensive trials and fewer resources. This reflects Hydro is more suitable for large-scale HPO jobs with limited resources, which is hard to handle by existing systems.

**Case II: multi-fidelity tuning.** When a user desires to obtain a good model with a relatively lower cost, multi-fidelity tuning is applied to search hyperparameters efficiently. Table 3 reports the Hydro performance on multi-fidelity tuning. We keep the same experiment settings as Case I, except using half GPU resources. Besides, we configure ASHA [63] with  $bracket = 1, grace = 3, reduction = 3$ . We observe that Hydro can achieve 5.8~33.4× reduction over Ray. Hydro can further benefit ASHA due to its much higher concurrency, which prevents the inaccurate promotion issue of ASHA [66]. Furthermore, we find that Hydro can also slightly improve the final model quality, which is mainly due to the different model initialization and more balanced layer-wise training rate configuration by Hydro parametrization. The results are also in line with Figure 1 that Hydro delivers a lower loss.

**Case III: tuning with a deadline.** When a user wants to get a model as good as possible by a fixed deadline, budget-bounded ASHA is applied. We simply evaluate two models with a deadline of 15 minutes as shown in Table 4. Hydro



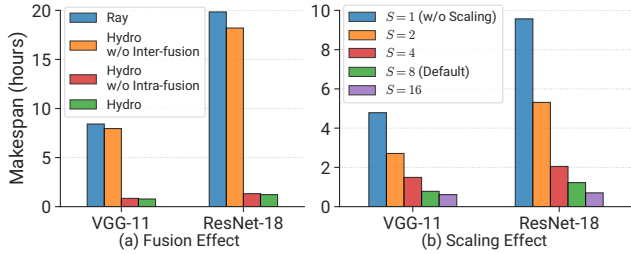


Figure 12: Ablation study. (a) Effect of inter- or intra-trial fusion. (b) Makespan of different scaling ratios.

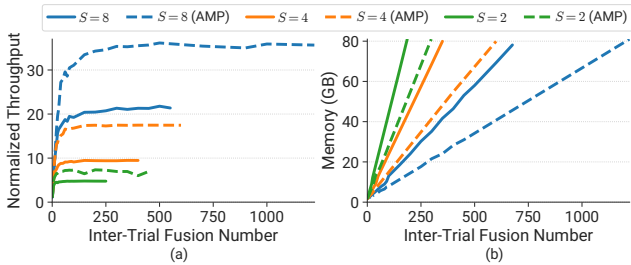


Figure 13: Sensitivity analysis of  $S$  and AMP on ResNet-18. (a) Accumulated throughput. (b) GPU memory footprint.

outperforms other baselines in final model accuracy within a limited time since it can well hide the target model training time inside the surrogate model tuning with Eager Transfer.

**End-to-end result visualization.** Figure 11 summarizes the makespan and accuracy of VGG-11 and ResNet-18 across different tuning algorithms and baselines. We note that Ray and Ray-ES share the same accuracy point since elastic and GPU sharing have no effect on the final model quality. The surrogate-based tuning (Hydro) can significantly reduce the search makespan without sacrificing the model accuracy. Due to the page limit, we only select these two models for presentation because of their relatively obvious efficacy of Ray-ES. Ray-ES has less improvement over Ray for larger models like WideResNet-50, since it cannot benefit from GPU sharing and the elastic improvement is limited (only for later stage).

## 6.4 More Evaluation on Hydro Tuner

**Ablation study of fusion.** Figure 12 (a) reveals an interesting observation that Hydro can only achieve very limited improvement over Ray if inter-trial fusion is disabled, even though we have scaled the model by  $8\times$ . This is because GPUs are underutilized for such small models and there is no evident training speedup although we scale the model. Hence, it is important to combine Model Shrinker and Trial Binder to achieve the desired performance. Additionally, we also evaluate the effect of intra-trial fusion. However, we find its improvement is limited on small models.

**Sensitivity analysis of scaling.** Figure 13 clearly presents the effect of the scaling ratio  $S$  on GPU memory and accumulated fused trial throughput, where the normalization base is the throughput of the target model. We find that the peak throughput increases linearly along with  $S$ . GPU memory also shows

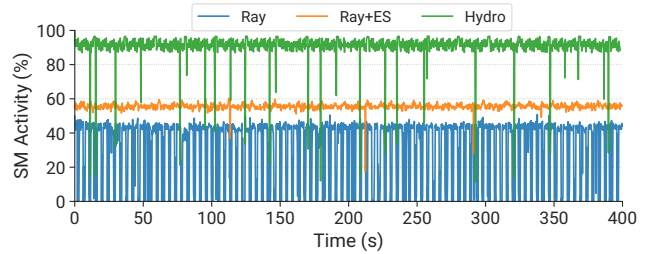


Figure 14: GPU utilization of HPO systems on ResNet-18.

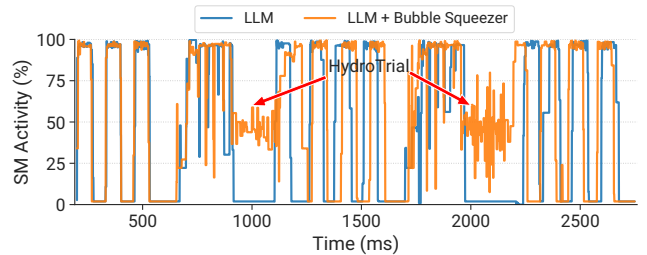


Figure 15: Visualizing Bubble Squeezer effect via DCGM. Two iterations of the first pipe stage are presented. The execution periods of the HydroTrial are highlighted by red arrows.

a similar pattern. In Figure 12 (b), we further evaluate the effect of the scaling ratio  $S$  on the overall tuning time. Hydro can continuously obtain benefits from higher scaling ratios. Besides, the final model accuracy maintains stable.

**Sensitivity analysis of AMP.** Figure 13 analyzes the effect of mixed precision training (i.e., AMP [15]), where solid and dashed lines represent the settings without and with AMP, respectively. We can find that the peak throughput can be further improved via enabling AMP. Besides, its effect on memory is also obvious, improving nearly  $2\times$  maximum fusion count.

**Impact on GPU utilization.** Figure 14 plots the GPU utilization traces on one GPU for 300 seconds using different HPO systems. We employ NVIDIA DCGM [11] to record SM Activity as GPU utilization. It is obvious that Hydro can achieve much higher GPU utilization than other baselines owing to the superior capability of inter-trial fusion [110].

**Overhead analysis.** We perform the overhead analysis on the ResNet-18 multi-fidelity tuning workload. Its overhead mainly derives from two aspects: (1) *profiling* accounts for 0.8%; (2) *defusion* (including trial restart) accounts for 3.3%. The associated overhead is minor when weighed against the substantial enhancements in the tuning efficiency of Hydro.

## 6.5 Hydro Coordinator Evaluation

**Bubble Squeezer.** To evaluate the impact of Bubble Squeezer, we interleave HydroTrials with a large GPT model over 32 A100 GPUs containing 4 pipeline stages on 4 nodes, which is implemented based on DeepSpeed [96] along with MegatronLM [56, 88, 102]. We measured the SM activity with and without Bubble Squeezer in Figure 15. Two traces are collected separately and we align them at the beginning of the figure. For the original GPT training, since the only active



kernel in the bubble is NCCL kernel for communication, the SM activity is extremely poor (about 2%) during the bubble. Hydro utilizes the unused SMs and achieves a relatively high SM utilization at about 50%, with no evident slowdown to the GPT model training. Here the HydroTrial is ResNet-18 model with fusion count  $F = 16$ , obtaining around 15% of exclusive throughput. We also measure the throughput influence of direct colocation and find it causes unacceptable interference (about 12% slowdown for the large model). Additionally, we further simulate the end-to-end performance of Bubble Squeezer. Here we set that the Hydro tuning job can only apply 1 exclusive GPU since most resources are occupied by the large model. We find the makespan of the tuning job can be greatly reduced by  $2.7\times$  with the free lunch.

**Heterogeneity-Aware Allocator.** We create a tiny cluster partition with 2 A100 and 2 V100 nodes (32 GPUs in total) to evaluate the impact of Heterogeneity-Aware Allocator. Besides, we uniformly sample 20 middle-size HPO jobs from Table 3 and randomly generate their job arrival time within one hour. Compared to resource-agnostic allocation, we find Heterogeneity-Aware Allocator achieves approximately a 1.3x reduction in the average job completion time.

## 7 Discussion

**Limitations.** Despite the extraordinary performance, Hydro’s surrogate-based tuning paradigm does have three limitations: (1) Hydro parametrization does not support regularization hyperparameters, such as weight decay and dropout, as elucidated in §4.1. (2) Hydro does not allow for any customized initialization techniques because Hydro implements its own automatic layer-wise re-initialization mechanism, which plays a crucial role in parameterization. (3) Hydro does not support fine-tuning since its theory is built atop i.i.d. samples (requiring the same dataset). Nevertheless, Hydro can deliver qualified models for most cases.

**Future work.** In the future, we plan to improve our work in following directions. (1) Supporting more DL frameworks like TensorFlow [19] and JAX [35]. (2) Considering more resource dimensions like CPU and network bandwidth besides GPU [83, 128], such as implementing the dataloader fusion of trials to further alleviate I/O contention. (3) Expanding the application scenarios such as cloud environments. It presents an opportunity for dynamic selection of heterogeneous spot instances, which can yield substantial cost savings [82, 106]. (4) Enabling partial model fusion across trails with minor architectural differences (e.g., add/remove/modify a few layers/blocks). Furthermore, Hydro can integrate model matching technique from ModelKeeper [60] to identify the models with similar architectures across jobs from different users and achieve cross-job level fusion, which can significantly improve cluster efficiency.

## 8 Related Work

**AutoML systems.** Automated Machine Learning (AutoML) refers to the process of automating the tasks associated with

optimizing ML model performance. In general, AutoML comprises two essential components: HPO and Neural Architecture Search (NAS). NAS systems (e.g., Retiarii [127], ModularNAS [74]) aim to discover the optimal model architecture for a specific task. On the other hand, HPO focuses on optimizing the hyperparameters of a fixed architecture, usually separate from NAS. Our work primarily concentrates on HPO.

Prior HPO systems like HyperSched [71], Rubberband [82] and Seer [32] support elastic training to allocate more GPU resources to promising trials, which is also supported in Hydro. Elastic training can make use of idle GPUs but fails to improve single GPU utilization. On the other hand, Fluid [125] further leverages NVIDIA MPS [13] technique to allocate multiple trials on a single GPU. HFTA [110] achieves inter-trial fusion on a shared accelerator. They can improve hardware utilization but only work well on tiny models (e.g., AlexNet [58], PointNet [93]). Based on the unique surrogate-base tuning nature, Hydro significantly extends the fusion application scope via model scaling and achieves automatic model fusion with minimum manual effort.

**Pipeline parallelism and interleaving execution.** Recent studies exploit bubbles induced by pipeline parallelism from multiple angles. Bamboo [106] fills redundant computations into bubbles to provide resilience and fast recovery for preemptible cloud instances. EnvPipe [26] selectively lowers the SM frequency of bubble periods to save energy. Unlike them, Hydro leverages bubbles to train HPO trials via interleaving execution, which is inspired by some prior works. For instance, Wavelet [109] and Zico [73] reduce the GPU peak memory based on interleaving. Muri [128] supports multi-resource interleaving to reduce contention.

## 9 Conclusion

This paper presents Hydro, a surrogate-based hyperparameter tuning service that provide job and cluster level optimization via automated model scaling, fusion and interleaving. Our experiments show that Hydro can dramatically reduce the tuning makespan and improve the cluster resource utilization.

## Acknowledgments

We sincerely thank our shepherd, Mathias Lécuyer, and the anonymous OSDI reviewers for their valuable comments on this paper. We also want to thank Greg Yang from Microsoft for the theory part support, Richard Liaw and Antoni Baum from Anyscale for the system development assistance, Shang Wang and Xin Li from UofT for their insightful discussion on inter-trial fusion, Shenggan Cheng and Shenggui Li from NUS for their constructive feedback on bubble squeezer. Additionally, we thank Li Ma and Shixin Yu for their technical support, as well as generous hardware resources from Shanghai AI Laboratory. This study is supported under the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s). Zhisheng Ye, Meng Zhang and Qiaoling Chen contribute equally to this work.

## References

- [1] Alibaba machine learning platform for ai. <https://www.alibabacloud.com/product/machine-learning>, 2023.
- [2] Amd epyc 7742 cpu. <https://www.amd.com/en/products/cpu/amd-epyc-7742>, 2023.
- [3] Chatgpt. <https://openai.com/blog/chatgpt>, 2023.
- [4] Duolingo. <https://www.duolingo.com/>, 2023.
- [5] Eleutherai gpt-neo 1.3b. <https://huggingface.co/EleutherAI/gpt-neo-1.3B>, 2023.
- [6] Github copilot. <https://github.com/features/copilot/>, 2023.
- [7] Infiniband networking. <https://www.nvidia.com/en-us/networking/products/infiniband/>, 2023.
- [8] Microsoft azure automated machine learning. <https://azure.microsoft.com/en-us/products/machine-learning/automatedml>, 2023.
- [9] Microsoft neural network intelligence. <https://github.com/microsoft/nni>, 2023.
- [10] Nvfuser. <https://github.com/pytorch/pytorch/projects/30>, 2023.
- [11] Nvidia data center gpu manager. <https://developer.nvidia.com/dcgm>, 2023.
- [12] Nvidia multi-instance gpu. <https://www.nvidia.com/en-us/technologies/multi-instance-gpu/>, 2023.
- [13] Nvidia multi-process service. <https://docs.nvidia.com/deploy/mps/index.html>, 2023.
- [14] Nvlink and nvswitch. <https://www.nvidia.com/en-us/data-center/nvlink/>, 2023.
- [15] Pytorch automatic mixed precision training. <https://pytorch.org/docs/stable/amp>, 2023.
- [16] Pytorch examples. <https://github.com/pytorch/examples>, 2023.
- [17] Pytorch fullyshardeddataparallel. <https://pytorch.org/docs/stable/fsdp>, 2023.
- [18] Torchvision new training recipe. <https://pytorch.org/blog/how-to-train-state-of-the-art-models-using-torchvision-latest-primitives/>, 2023.
- [19] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016.
- [20] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, 2016.
- [21] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations*, ICLR '15, 2015.
- [22] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.
- [23] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics — Workshop on Challenges & Perspectives in Creating Large Language Models*, ACL-BigScience '22, 2022.
- [24] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.
- [25] Lénaïc Chizat and Francis Bach. On the global convergence of gradient descent for over-parameterized models using optimal transport. In *Advances in Neural Information Processing Systems*, NeurIPS '18, 2018.
- [26] Sangjin Choi, Inho Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. Envpipe: Performance-preserving dnn training framework for saving en-

ergy. In *2023 USENIX Annual Technical Conference, USENIX ATC '23*, 2023.

- [27] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, 2022.
- [28] Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rouesnel, Philip Gautier, Zohar Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, Wilton Wu, Aida Zolic, Huibin Shen, Amr Ahmed, Fela Winkelmolen, Miroslav Miladinovic, Cedric Archambeau, Alex Tang, Bhaskar Dutt, Patricia Grao, and Kumar Venkateswar. Amazon sagemaker autopilot: a white box automl solution at scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning, DEEM '20*, 2020.
- [29] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.
- [30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics, NAACL '19*, 2019.
- [31] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI '15*, 2015.
- [32] Lisa Dunlap, Kirthevasan Kandasamy, Ujval Misra, Richard Liaw, Michael Jordan, Ion Stoica, and Joseph E. Gonzalez. Elastic hyperparameter tuning on the cloud. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC '21*, 2021.
- [33] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML '18*, 2018.
- [34] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '21*, 2021.
- [35] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *Proceedings of Machine Learning and Systems, MLSys '18*, 2018.
- [36] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *CoRR*, 2022.
- [37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, ICML '10*, 2010.
- [38] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. <https://skylion007.github.io/OpenWebTextCorpus/>, 2023.
- [39] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17*, 2017.
- [40] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI '19*, 2019.

- [41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR '16*, 2016.
- [42] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling laws for autoregressive generative modeling. *CoRR*, 2020.
- [43] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack William Rae, and Laurent Sifre. An empirical analysis of compute-optimal large language model training. In *Advances in Neural Information Processing Systems, NeurIPS '22*, 2022.
- [44] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision, ICCV '19*, 2019.
- [45] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, 2021.
- [46] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '23*, 2023.
- [47] Yimin Huang, Yujun Li, Hanrong Ye, Zhenguo Li, and Zhihua Zhang. Improving model training with multi-fidelity hyperparameter evaluation. In *Proceedings of Machine Learning and Systems, MLSys '22*, 2022.
- [48] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML '15*, 2015.
- [49] Arthur Jacot, Franck Gabriel, and Clement Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in Neural Information Processing Systems, NeurIPS '18*, 2018.
- [50] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference, USENIX ATC '19*, 2019.
- [51] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, 2019.
- [52] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, 2020.
- [53] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on International Conference on Machine Learning, ICML '13*, 2013.
- [54] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations, ICLR '15*, 2015.
- [55] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations, ICLR '17*, 2017.
- [56] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *CoRR*, 2022.
- [57] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.
- [58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems, NeurIPS '12*, 2012.
- [59] Sameer Kumar, Yu Wang, Cliff Young, James Bradbury, Naveen Kumar, Dehao Chen, and Andy Swing. Exploring the limits of concurrency in ml training on google tpus. In *Proceedings of Machine Learning and Systems, MLSys '21*, 2021.



- [60] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. ModelKeeper: Accelerating DNN training via automated training warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, 2023.
- [61] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *CoRR*, 2022.
- [62] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.
- [63] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Proceedings of Machine Learning and Systems*, MLSys '20, 2020.
- [64] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 2018.
- [65] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.
- [66] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Jixiang Li, Ji Liu, Ce Zhang, and Bin Cui. Hyper-tune: towards efficient hyper-parameter tuning at scale. *Proceedings of the VLDB Endowment*, 2022.
- [67] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Zhi Yang, Ce Zhang, and Bin Cui. Transbo: Hyperparameter optimization via two-phase transfer learning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, 2022.
- [68] Yang Li, Yu Shen, Jiawei Jiang, Jinyang Gao, Ce Zhang, and Bin Cui. Mfes-hb: Efficient hyperband with multi-fidelity quality measurements. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '21, 2021.
- [69] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, and Bin Cui. Openbox: A generalized black-box optimization service. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, 2021.
- [70] Yang Li, Yu Shen, Wentao Zhang, Jiawei Jiang, Bolin Ding, Yaliang Li, Jingren Zhou, Zhi Yang, Wentao Wu, Ce Zhang, and Bin Cui. Volcanoml: speeding up end-to-end automl via scalable search space decomposition. *Proceedings of the VLDB Endowment*, 2021.
- [71] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, 2019.
- [72] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *CoRR*, 2018.
- [73] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference*, USENIX ATC '21, 2021.
- [74] Yunfeng Lin, Guilin Li, Xing Zhang, Weinan Zhang, Bo Chen, Ruiming Tang, Zhenguo Li, Jiashi Feng, and Yong Yu. Modularnas: Towards modularized and reusable neural architecture search. In *Proceedings of Machine Learning and Systems*, MLSys '21, 2021.
- [75] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *CoRR*, 2019.
- [76] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV '21, 2021.
- [77] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, ICLR '19, 2019.
- [78] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, 2020.
- [79] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 2014.

- [80] Song Mei, Andrea Montanari, and Phan-Minh Nguyen. A mean field view of the landscape of two-layer neural networks. *Proceedings of the National Academy of Sciences*, 2018.
- [81] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations, ICLR '17*, 2017.
- [82] Ujval Misra, Richard Liaw, Lisa Dunlap, Romil Bhardwaj, Kirthevasan Kandasamy, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Rubberband: Cloud-based hyperparameter tuning. In *Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21*, 2021.
- [83] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI '22*, 2022.
- [84] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elilbol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI '18*, 2018.
- [85] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, 2019.
- [86] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *Proceedings of the 38th International Conference on Machine Learning, ICML '21*, 2021.
- [87] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20*, 2020.
- [88] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, 2021.
- [89] Radford M. Neal. *Priors for Infinite Networks*. Springer New York, 1996.
- [90] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 2008.
- [91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems, NeurIPS '19*, 2019.
- [92] Valerio Perrone, Huibin Shen, Aida Zolic, Iaroslav Shcherbatyi, Amr Ahmed, Tanya Bansal, Michele Donini, Fela Winkelmolen, Rodolphe Jenatton, Jean Baptiste Faddoul, Barbara Pogorzelska, Miroslav Miladinovic, Krishnaram Kenthapadi, Matthias Seeger, and Cédric Archambeau. Amazon sagemaker automatic model tuning: Scalable gradient-free optimization. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, KDD '21*, 2021.
- [93] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR '17*, 2017.
- [94] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI '21*, 2021.
- [95] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [96] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20*, 2020.

- [97] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. torch.fx: Practical program capture and transformation for deep learning in python. In *Proceedings of Machine Learning and Systems, ML-Sys '22*, 2022.
- [98] Daniel A. Roberts, Sho Yaida, and Boris Hanin. *The Principles of Deep Learning Theory*. Cambridge University Press, 2022.
- [99] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, 2017.
- [100] David Salinas, Matthias Seeger, Aaron Klein, Valerio Perrone, Martin Wistuba, and Cedric Archambeau. Syne tune: A library for large scale hyperparameter tuning and reproducible research. In *First Conference on Automated Machine Learning, AutoML '22*, 2022.
- [101] Noam Shazeer, \*Azalia Mirhoseini, \*Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations, ICLR '17*, 2017.
- [102] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, 2020.
- [103] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations, ICLR '15*, 2015.
- [104] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems, NeurIPS '12*, 2012.
- [105] Xingyou Song, Sagi Perel, Chansoo Lee, Greg Kochanski, and Daniel Golovin. Open source vizier: Distributed infrastructure and api for reliable and flexible blackbox optimization. In *First Conference on Automated Machine Learning, AutoML '22*, 2022.
- [106] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI '23*, 2023.
- [107] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI '22*, 2022.
- [108] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems, NeurIPS '17*, 2017.
- [109] Guanhua Wang, Kehan Wang, Kenan Jiang, XI-ANGJUN LI, and Ion Stoica. Wavelet: Efficient dnn training with tick-tock scheduling. In *Proceedings of Machine Learning and Systems, MLSys '21*, 2021.
- [110] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. In *Proceedings of Machine Learning and Systems, MLSys '21*, 2021.
- [111] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI '22*, 2022.
- [112] William F. Whitney. Parallelizing neural networks on one gpu with jax, 2023.
- [113] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP '20*, 2020.
- [114] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI '18*, 2018.
- [115] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on*

*Operating Systems Design and Implementation*, OSDI '20, 2020.

- [116] Sho Yaida. Meta-principled family of hyperparameter scaling strategies. *CoRR*, 2022.
- [117] Ge Yang, Edward Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tuning large neural networks via zero-shot hyperparameter transfer. In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021.
- [118] Greg Yang. Wide feedforward or recurrent neural networks of any architecture are gaussian processes. In *Advances in Neural Information Processing Systems*, NeurIPS '19, 2019.
- [119] Greg Yang. Tensor programs ii: Neural tangent kernel for any architecture. *CoRR*, 2020.
- [120] Greg Yang. Tensor programs iii: Neural matrix laws. *CoRR*, 2021.
- [121] Greg Yang and Edward J. Hu. Tensor programs iv: Feature learning in infinite-width neural networks. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, 2021.
- [122] Greg Yang and Etai Littwin. Tensor programs iib: Architectural universality of neural tangent kernel training dynamics. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, 2021.
- [123] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. Astraea: A fair deep learning scheduler for multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2021.
- [124] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems*, ML-Sys '20, 2020.
- [125] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In *Proceedings of Machine Learning and Systems*, ML-Sys '21, 2021.
- [126] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *British Machine Vision Conference*, BMVC '16, 2016.
- [127] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarrii: A deep learning Exploratory-Training framework. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020.
- [128] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '22, 2022.
- [129] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020.
- [130] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, 2022.







# MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms

Yuke Wang, Boyuan Feng, Zheng Wang, <sup>†</sup>Tong Geng, \*Kevin Barker, \*Ang Li, and Yufei Ding  
<sup>†</sup>University of Rochester, \*Pacific Northwest National Laboratory  
University of California, Santa Barbara

## Abstract

The increasing size of input graphs for graph neural networks (GNNs) highlights the demand for using multi-GPU platforms. However, existing multi-GPU GNN systems optimize the computation and communication individually based on the conventional practice of scaling dense DNNs. For irregularly sparse and fine-grained GNN workloads, such solutions miss the opportunity to jointly schedule/optimize the computation and communication operations for high-performance delivery.

To this end, we propose **MGG**, a novel system design to accelerate full-graph GNNs on multi-GPU platforms. The core of MGG is its novel dynamic software pipeline to facilitate fine-grained computation-communication overlapping within a GPU kernel. Specifically, MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping. MGG also incorporates an intelligent runtime design with analytical modeling and optimization heuristics to dynamically improve the execution performance. Extensive evaluation reveals that MGG outperforms state-of-the-art full-graph GNN systems across various settings: on average  $4.41\times$ ,  $4.81\times$ , and  $10.83\times$  faster than DGL, MGG-UVM, and ROC, respectively.

## 1 Introduction

Over the recent years, graph-based deep learning has attracted lots of attention from the research and industry communities. Among various graph-learning methods, graph neural network (GNN) [21, 43, 49] gets highlighted most due to its success in many deep learning tasks (e.g., node feature vector (embedding) generation for node classification [11, 13, 19] and link prediction [7, 22, 42]). GNNs consist of several layers, where layer  $k+1$  computes the embedding for a node  $v$  based on the embeddings at the previous layer  $k$  ( $k \geq 0$ ) by applying

$$a_v^{(k+1)} = \mathbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)})$$
$$h_v^{(k+1)} = \mathbf{Update}^{(k+1)}(a_v^{(k+1)})$$

where  $h_v^{(k)}$  is the embedding of node  $v$  at layer  $k$ . The *Aggregate* function accumulates neighbors'  $\mathbf{N}(v)$  embeddings of node  $v$ . The *Update* function consists of a fully-connected NN layer. The neighbor aggregation (*Aggregate*) is the key bottleneck that dominates the overall computation due to its high computation sparsity and irregularity [46, 50]. Compared with conventional graph analytics (e.g., random walk [14, 39]), GNN features higher accuracy [21, 49] and better generality [16, 55] on various applications.

GNN computation on large input graphs (millions/billions of nodes and edges) usually counts on powerful multi-GPU platforms (e.g., NVIDIA DGX [35]) for scaling up the performance. The multi-GPU system (that can potentially store all data required for the computation in the aggregate memory of all GPUs on a single machine) can benefit from aggregated memory capacity and bandwidth (HBM and NVLinks) with more GPUs. There is also a popular trend for state-of-the-art hyper-scale systems employing GPU-centric building blocks. For example, the recent NVIDIA DGX SuperPod [33] consists of  $32 \times$  DGX-H100 servers (each with  $8 \times$  H100). Unfortunately, the runtime performance of GNNs does not scale proportionally with the aggregated compute capability and memory capacity of the platform. This is mainly because the irregular and sparse local memory access of neighbor aggregation in the single-GPU settings now “scales” to more expensive inter-GPU communication (i.e., remote memory access). Such intensive inter-GPU communication becomes the new critical path of multi-GPU GNN execution and offsets the performance gains from multi-GPU computation parallelism.

Based on this observation, we highlight a more promising way of formalizing GNN computation on multi-GPU systems. Our key insight is that GNN execution can be more precisely abstracted as a fine-grained dynamic software pipeline to encourage communication and computation overlapping, which will largely hide the communication cost. The opportunities for building such fine-grained pipelines widely exist at different granularities in GNNs. For instance, on a single graph node, the remote neighbor access can be overlapped with the local neighbor computation. Among different graph nodes,

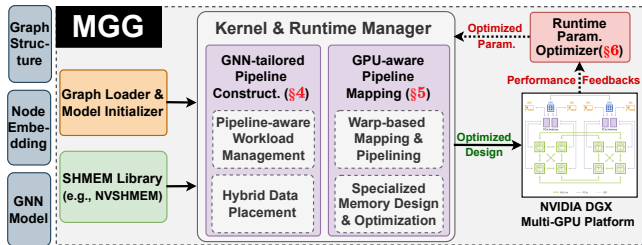


Figure 1: Overview of MGG.

the remote neighbor access for certain nodes would potentially be overlapped with the local neighbor computation of some other nodes. However, prior research could hardly exploit such benefits since they rely on hardware and software infrastructures tailored for coarse-grained [18, 28] and regular communication patterns [12, 26]. To capitalize on the fine-grained pipelining benefits, there are three major challenges.

The first challenge is *how to craft the pipeline structure*. A work-efficient pipeline for GNNs demands comprehensively considering multiple factors (e.g., the operations and the number/granularity of each pipeline stage) to best fit the GNN algorithm and multi-GPU computation/communication. The second challenge is *how to map the pipeline to the GPU processing units*. Given the GPU’s architectural complexity (e.g., multi-granular processing units and multi-layer memory hierarchy), different mapping and primitive choices would bring performance and design flexibility tradeoffs. The third challenge is *how to find and adapt toward the “optimal” pipeline configuration swiftly*. Given the diversity of GNN inputs (e.g., graph structures) and hardware (e.g., different types/numbers of GPUs), pinpointing the best-off design configuration with high-performance delivery relies on combined insights from the properties of the software pipeline, GNN inputs, and GPU programming and execution paradigms.

To this end, we introduce a set of principles for multi-GPU GNN acceleration via a fine-grained dynamic software pipeline. *To construct fine-grained pipelines*, the original coarse-grained irregular GNN computation should be broken-down into fine-grained operations. The joint optimization of the GNN workload granularity and data layout should be carried out to facilitate operation overlapping. *To map pipelines to GPUs*, the proper GPU logical processing units (e.g., thread, warp, and block) should be selected for promoting GPU kernel efficiency and design flexibility. In addition, the right choice of communication primitives (e.g., NVSHMEM [36]) should be determined to provide fine-grained inter-GPU communication support. *To adapt pipelines dynamically*, customized kernel templates with tuning knobs should be devised. This will help to maintain pipelining effectiveness across a diverse range of GNN inputs and hardware platform settings.

We crystallize the above principles into MGG<sup>1</sup>, a holistic system design and implementation for multi-GPU GNNs (Figure 1). Given the GNN models and inputs, MGG will automat-

ically generate pipeline-centric GPU kernels for multi-GPU platforms and dynamically improve the kernel performance based on runtime feedback. The core of MGG is its **Kernel & Runtime Manager**, which constructs GNN-tailored pipelines and maps such pipelines to proper communication primitives and GPU logical processing units. It can also dynamically orchestrate GPU kernels based on new configurations. MGG also incorporates a **Runtime Parameter Optimizer**, which will monitor the performance (e.g., latency) from the actual execution and generate new configurations for the next iteration based on the analytical performance model and optimization heuristics. To the best of our knowledge, we are the first to explore the potential of GPU kernel operation pipelining for accelerating irregular GNN workloads. Moreover, MGG can be generalized to other applications (e.g., deep-learning recommendation model (DLRM) [31]) that are sharing similar irregular communication demands (§7.3).

Overall, we make the following contributions in this paper:

- We propose a GNN-tailored pipeline construction technique (§4) with pipeline-aware workload management and hybrid data placement, for efficient communication-computation pipelining in a GPU kernel.
- We introduce a GPU-aware pipeline mapping strategy (§5), encompassing warp-based mapping and pipelining, and specialized memory designs and optimizations to comprehensively promote kernel performance.
- We devise an intelligent runtime with lightweight analytical modeling and optimization heuristics to dynamically improve the performance of GNN training (§6).
- Comprehensive experiments demonstrate that MGG can outperform state-of-the-art multi-GPU GNN systems across various GNN benchmarks. Additionally, MGG can be generalized to other DL applications, like DLRM.

## 2 Related Work

Recent deep-learning applications expand their scope from handling structured dense inputs (e.g., images) to unstructured sparse inputs (e.g., graphs). Along with such algorithmic/application expansion is the exploration of new system designs and optimizations for more efficient deep learning. One of the most important topics is the ability to handle large-scale inputs, which are usually out of the computation and memory capacity of one GPU. For scaling regular deep-learning applications, like dense DNNs, various abstractions (e.g., data and model parallel) and high-performance communication libraries (e.g., NCCL [34]) have been developed. While the scaling approach for irregular GNN applications is still initial and suffers from unsatisfactory performance.

Compared to scaling dense DNNs, scaling sparse GNNs is significantly more challenging. The irregular fine-grained sparse GNNs workload cannot fit the regular coarse-grained

<sup>1</sup><https://github.com/YukeWang96/MGG-OSDI23-AE.git>

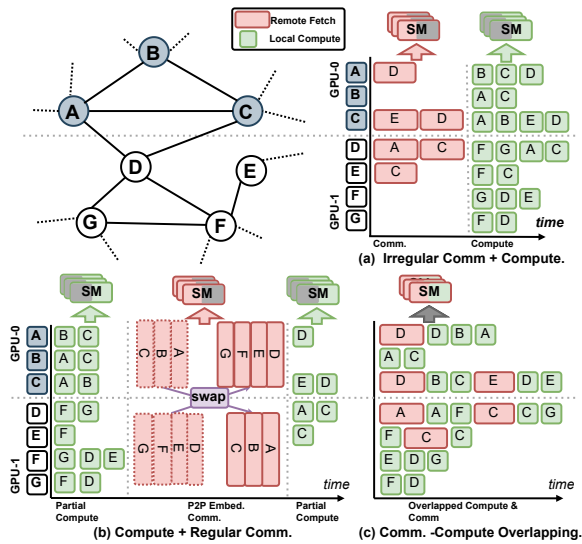


Figure 2: Different Multi-GPU GNN strategies for computation and communication. Note that red and green boxes indicate aggregation workload on remote and local neighbors. “SM” boxes with grey areas indicate potential idleness.

workload abstraction for dense DNNs. The cost of irregular communication in GNNs cannot be easily amortized by simply batching more requests as dense DNNs due to their randomness and sparseness. Scaling strategies largely vary among different GNN inputs while tiling/schedule strategies would be reused across different inputs of dense DNNs. Therefore, an array of dedicated designs have been introduced to scale the sparse GNNs, focusing on three major directions.

**Operator Specialization for Sparse Communication:**

This is the mainstream solution that treats the communication as a standalone operator for irregularly sparse GNN communication (Figure 2(a)). DGL [45] is the state-of-the-art GNN framework and its most recent update incorporates PyTorch-Direct [28] (a GNN-tailored communication design based on zero-copy memory [41]) for large-scale GNN training across GPUs. Work from [6] introduces a communication planning algorithm for distributed GNNs by considering links, communication, contention, and load balancing. However, these efforts optimize the communication standalone and thus miss the opportunities to jointly optimize computation and communication operations/schedules which can potentially reduce the overall latency and improve GPU utilization.

**Algorithm Modification for no Communication:**

The second typical type is to eliminate irregular communication by altering algorithms [25, 45, 51]. They harness various algorithmic adaption solutions, such as neighbor sampling and mini-batch to prefetch the remote neighbors to local devices, and then train the GNN model in a data-parallel fashion as the traditional dense DNN. However, existing research [8, 18] shows that such an algorithmic modification would compromise the accuracy of GNN models compared to the original

GNNs. It would also destabilize the algorithmic performance (e.g., the lower convergence speed and final accuracy) under different inputs and sampling configurations.

**Schedule Transformation for Dense Communication:**

The third type is to transform irregular communication to regularized communication (e.g., AlltoAll, P2P), which has been optimized by existing communication kernels (Figure 2(b)). ROC [18] delegates communication to its underlying NVIDIA Legion runtime [5], which manages irregular remote neighbor access via a DMA engine. It batches fine-grained embeddings into large embedding tiles on CPUs to facilitate coarse-grained data movement between the host and GPUs. NeuGraph [26] tiles the large node embedding matrices by rows (as embedding chunks) and then forwards each chunk to GPUs sequentially via coarse-grained P2P communication. P3 [12] spots the potential of transforming irregular embedding communication to regular all-to-all communication for embedding column tiles. However, this type of effort would introduce many unnecessary data movements and non-trivial overhead to transform original algorithms and data inputs.

To sum up, existing designs explore solutions in a limited scope and have yet to extend their solution search to a broader context by exploring the synergy between the multi-GPU GNN workloads, GPU execution paradigms, and communication patterns. Therefore, these designs could hardly enjoy the full potential of multi-GPU platforms.

**3 Motivation**

Different from prior solutions, we propose a new view for multi-GPU GNN workload. We spot that by removing the explicit barrier between the computation and communication stage in multi-GPU GNNs, we can co-schedule the operations from both stages in a holistic way that can reduce the GPU resource idleness and promote performance (Figure 2(c)). For example, when GPUs initiate remote access requests and are waiting for the arrival of remote data, the idle cycles of GPUs can be fulfilled by other local computing workloads. Such insight enables us to abstract the multi-GPU GNN workload as a fine-grained dynamic software pipeline for communication and communication overlapping. Specifically, “Fine-grained” means that the operations at each pipeline stage are tiny (e.g., the aggregation of one neighbor’s embeddings) versus DNN layers. “Dynamic” means that the division of computation into pipeline stages would vary among different inputs in contrast to DNNs with a relatively fixed pipeline. Such a new design is motivated by our three major observations.

**GNN Workload Speciality:**

The first observation reveals the specialty of GNN workloads, which feature two major types of partial dependency that facilitate pipelining [1]. The first type is the fine-grained neighbor aggregation dependency, where the neighbor embeddings of individual graph nodes are aggregated either sequentially or in parallel with proper synchronization. The second type is the dynamic execution



Listing 1: NVSHMEM APIs in CUDA C.

```

1 // Initialize an NVSHMEM context on CPUs.
2 nvshmem_init();
3 // Get the current GPU device ID on CPUs.
4 int gpu_id = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
5 // Set the GPU based on its device ID on CPUs.
6 cudaSetDevice(gpu_id);
7 // Define NVSHMEM memory visible for all GPUs on CPUs.
8 d_shared_mem = (void*) nvshmem_malloc (num_bytes);
9 // Define global memory visible only for the current GPU.
10 cudaMalloc((void**) &d_mem, num_bytes);
11 // Remote access API called by a thread/warp/block.
12 __device__ nvshmem_float_get_{warp/block}(void *dst, const
    void *src, size_t nelems, int src_gpu_id);
13 // Sync all GPUs within an NVSHMEM context on CPUs.
14 nvshmem_barrier_all();
15 // Release NVSHMEM objects on CPUs.
16 nvshmem_free(d_shared_mem);
17 // Terminate the current NVSHMEM context on CPUs.
18 nvshmem_finalize();

```

dependency on limited processing units, where different operations would compete for limited GPU resources (e.g., SMs) during the runtime. Such two types of dependencies expose new opportunities for us to amortize communication costs by overlapping neighbor aggregation from different nodes.

**GPU Execution Characteristics:** The second observation highlights the characteristics of the GPU execution paradigm. One key design principle of GPUs is their massive computation/communication parallelism to amortize the unit cost of individual computation/communication operations [40]. The underlying mechanism of GPU hardware design to facilitate this is to simultaneously schedule multiple logical processing units (e.g., threads/warps/blocks) to share the hardware processing units (i.e., GPU SMs). Such a design provides the essential ingredient for pipelining, which is that computation and communication operations can co-run on the same units at the same time to fulfill the idle GPU cycles and maximize the utilization of the GPU hardware processing units. Moreover, with the precise control of GPU kernel launching parameters (e.g., the size of the block and shared memory), the effectiveness of co-running heterogeneous operations can be adjusted so that we can flexibly accommodate different inputs while maintaining high-performance delivery.

**Multi-GPU Programming Support:** The third observation features the recent advancement of the GPU communication technique and its programming support. The one highlighted most is the NVSHMEM [36], which provides GPU intra-kernel APIs for fine-grained (several to tens of bytes) inter-GPU communication (Listing 1). NVSHMEM is the main communication backend for MGG. Other existing techniques such as Zero-copy memory can also serve as an alternative to NVSHMEM for fine-grained communication. The performance will be similar while NVSHMEM offers better programmability. Some other traditional strategies for inter-GPU communication, would either offer too

coarse-grained communication solutions (e.g., unified virtual memory [38] uses KB-level communication granularity) or resort to the default communication strategies of existing multi-GPU-based runtime system (e.g., NVIDIA Legion [5]) without GNN-tailored communication optimization.

These observations and insights motivate MGG, a holistic multi-GPU GNN system with a novel view of GNN workloads as an operation pipeline. MGG automates the pipeline construction, detailed pipeline mapping, and dynamic input-driven pipeline adaption, to improve the GNN scaling.

## 4 GNN-tailored Pipeline Construction

Constructing a GNN-tailored pipeline are facing two major challenges: 1) *How to effectively partition and schedule multi-GPU GNN workloads* so that pipeline efficiency can be maximized; 2) *How to properly layout input* so that the hierarchy of GNN inputs and the memory/storage of multi-GPU systems can be carefully matched to facilitate pipeline execution. MGG addresses these challenges with *Pipeline-aware Workload Management* and *Hybrid GNN Data Placement*.

### 4.1 Pipeline-aware Workload Management

Managing irregularly sparse GNN workloads for pipelining is challenging and could hardly benefit from the prior practice and exploration of the DNN pipeline [29, 30].

**Difference from DNN pipeline** *First*, balancing the GNN workloads among GPUs has to jointly optimize the computation capacity and the computation/communication irregularity. While the DNN pipeline only needs to balance the computation/memory capacity, since its pipeline stages are well-structured and their inputs are regularly dense. Distributed DNNs require dense regular communication (e.g., Allreduce) that is naturally fit for existing GPU interconnects optimized for throughput and has been optimized by many libraries (e.g., NCCL). In contrast, distributed full-graph GNN (with the entire graph cached on GPUs) is much more challenging since it requires sparse irregular communication that is naturally at odds with the existing hardware interconnects, and fewer efforts have optimized its performance. *Second*, the GNN pipeline workload is more irregular and non-structural and can easily cause pipeline stalls/bubbles. For example, remote neighbor aggregation would have different stages (remote access + aggregation) compared with local neighbor aggregation (local access + aggregation), making it challenging to mix those two heterogeneous workloads. While in the DNN pipeline, all inputs should consistently pass through the same pipeline stages. *Third*, GNN pipeline stages are more fine-grained (e.g., fetching individual embeddings) compared with coarse-grained layers (e.g., GEMMs and Convolutions) in the DNN pipeline. Such small workload granularity enables different pipeline stages to overlap with each other on GPU processing units, like Streaming Multiprocessors (SMs). In

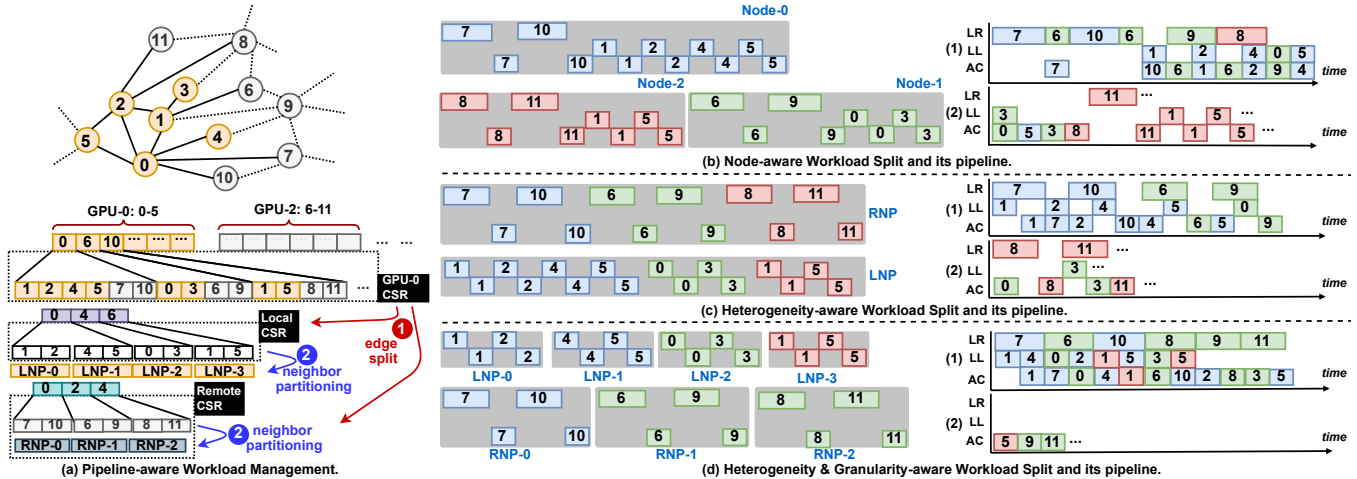


Figure 3: (a) Pipeline-aware workload management. “LNP”/“RNP” indicate local/remote workload partitions. (b)(c)(d) Different strategies of workload decomposition and pipelining. Each box indicates a certain (local/remote) aggregation workload and its length indicates its relative latency. “LR”: loading remote neighbors, “LL”: loading local neighbors, “AC”: aggregation computation. Each grey rectangular shadow indicates a workload partition to be processed by one GPU processing unit. (1) and (2) indicate that the same pipeline is chunked into two parts along its time axis due to space limitations.

contrast, DNN pipelines can only overlap layer-wise computation and communication operations among different GPUs.

With the above insights, we propose a three-stage dynamic software pipeline design. The three stages include *loading remote neighbors (LR)*, *loading local neighbors (LL)*, and *aggregation computation (AC)*. Aggregation of a certain neighbor will only take two stages. The remote neighbor aggregation will take the stage LR and AC while local neighbor aggregation will take the stage LL and AC. The stage-wise pipelining is achieved with two steps: 1) assigning aggregation workload to different GPU logical processing units (LPUs), like warps and blocks, and 2) scheduling different LPUs on the same GPU SM to overlap their execution. Three-phase pipeline can generalize to different GNN models, which essentially consist of the different numbers of basic remote and local operations. For example, GCN has a lower local-versus-remote operation ratio while GAT features a higher local-versus-remote operation ratio. Three-phase pipeline can also capture differences among inputs. For instance, a more sparse graph will have a higher remote-to-local operation ratio.

However, the direct construction and execution of such three-stage pipelines would be inefficient, because of its ignorance of GNN workload heterogeneity and irregularity on multi-GPU platforms. To address these challenges, MGG highlights a GNN-tailored pipeline construction strategy to build and optimize the software pipeline in three steps.

**Step-1: Workload-aware inter-GPU pipeline workload balancing.** This step aims to construct the “raw” pipeline and balance workloads among pipelines on different GPUs. Our insight is that GPUs with massive processing units (e.g., SMs) will serve many pipelines concurrently, and the key to maximizing GPU performance and utilization is to en-

sure that each pipeline will get a similar amount of workload, thereby avoiding execution critical path on certain “long” pipelines. We, therefore, develop a *range-constrained binary search algorithm* (Algorithm 1) based on prior graph partitioning exploration [3]. Our solution features a lower runtime cost to split the GNN input graph into chunks (one chunk per GPU) while balancing the number of edges within each chunk. Then the workload from the same chunk is grouped by nodes as *workload partitions* mixed local and remote neighbors (Figure 3(b)). From its potential execution pipeline, we can see many idle cycles (indicated by blank spaces in different pipeline stages) which would result in low pipeline efficiency and GPU resource occupancy. Note that in the software pipeline, workloads from different partitions can be overlapped as they will be processed by different LPUs. While the workloads from the same partition are sequentially processed by one LPU and their relative order should be maintained even after being mixed with other partitions.

**Step-2: Heterogeneity-aware pipeline bubble reduction.** The pipeline constructed from the previous step is still inefficient due to its scattered workloads among stages, namely pipeline bubbles. The optimization in this step is to minimize such pipeline bubbles for better pipeline efficiency. The key is to reduce the heterogeneity of workload partitions that hinders effective overlapping. To achieve this, we categorize the sparse multi-GPU GNN computation into two types. The first type has local neighbor access only, which has shorter execution latency. The second type has remote neighbor access, which features high latency overhead. We delicately handle different types of workloads via grouping (Figure 3(a)-1), where two separate CSRs for *local* and *remote* subgraphs will be built. The aggregation will be conducted on local and

---

**Algorithm 1:** Range-constrained Binary Search.

---

```
input :Graph node pointer array (nPtr), edge list array
        (eList), and the number of GPUs (numGPUs).
output :list of graph edge split points (numGPUs - 1).
1  outList = {};
2  lastPos = 0;
   /* Compute approximated #edges per GPU.          */
3  ePerGPU = (len(eList) + numGPUs - 1) / numGPUs;
4  for sId in [0, 1, ..., numGPUs - 1] do
5  |   nid = binSearch(nPtr, ePerGPU, lastPos, numNodes);
6  |   lastPos = nid;
7  |   outList[sId] = nid;
8  end
9  return outList;

   /* Search split points on nPtr.                  */
10 Function binSearch(nPtr, ePerGPU, lastPos,
    numNodes):
11 |   i = lastPos;
12 |   j = numNodes;
13 |   target = min(nPtr[i] + ePerGPU, nPtr[numNodes]);
14 |   while i < j do
15 | |   mid = (nPtr[i] + nPtr[j]) / 2;
16 | |   if mid > target then
17 | | |   j = (i + j) / 2;
18 | | |   else
19 | | |   i = (i + j) / 2;
20 |   end
21 |   return i;
```

---

remote subgraphs separately and followed by a result synchronization at the end. Such a remote-local split is also backed by the fact that on platforms with all-to-all GPU interconnections (e.g., DGX-A100/H100), accessing different GPUs under the same data granularity has approximately equal communication cost [24]. Such heterogeneity awareness in workload partitioning (Figure 3(c)) enables a more densely overlapped workload between the stage LR and LL/AC.

**Step-3: Granularity-aware intra-GPU pipeline enhancement.** While the second optimization improves pipeline efficiency by reducing the workload heterogeneity, there is still plenty of room for further enhancement. The optimization in this step is to facilitate a more balanced workload distribution among pipeline stages. This key is to find the proper workload granularity for local and remote subgraphs so that those originally sequentially processed workload partitions can be overlapped. Our key observation is that nodes in the local/remote subgraphs would have a diverse number of neighbors. Such a specialty makes it challenging for massively parallel GPUs to harvest the real performance gains due to the imbalance workload and diverged execution flow. Therefore, we approximate such coarse-grained irregular workloads with fine-grained fixed-sized partitions so that the workload imbalance across nodes can be amortized. For example, with 2 neighbors per partition (Figure 3(a)-②), we can get a more balanced workload among nodes in their local and remote

neighbor aggregation. With such granularity awareness, the individual pipeline can be further condensed along its time axis with more overlapping of the LL and AC stage. (Figure 3(d)). Meanwhile, the irregular workload can be more evenly distributed to GPU SMs for higher GPU utilization. On the other side, partition granularity should also be balanced with synchronization overhead, since more fine-grained partitioning can bring more parallelism at the cost of more synchronization overhead. This is because workloads from different partitions for the same target node need to be reduced via synchronization, like inter-thread shuffling and atomics.

MGG design can also be generalized to multiple machines with a minor adaptation. For example, in Figure 3(d), when there are inter-node (over Inifite-Band) remote neighbors (longer latency due to lower inter-node communication speed), the size of remote neighbor partitioning (RNP) should be adjusted to a smaller size (e.g., from 2 to 1 remote neighbor) to facilitate better overlapping with local computation.

## 4.2 Hybrid GNN Data Placement

In collaboration with our multi-step pipeline construction, we introduce a *hybrid GNN data placement* strategy to exploit the benefits of different types of memory in SHMEM-enabled multi-GPU systems. The major impact of such hybrid placement on pipelining is two-fold. First, placing GNN data in different memory spaces will lead to different ratios of local and remote workloads, thus, affecting workload balance among pipelines. Second, different memory spaces will offer different access performances (e.g., latency), thereby, affecting the execution efficiency of the individual pipelines, such as the number of pipeline bubbles.

Our strategy focuses on two major aspects. Firstly, for workload balance among pipelines, we leverage NVSHMEM “shared” global memory to store the node embeddings (NEs) of the whole graph (Figure 4 left). Our major consideration here is that such shared global memory space can be accessed by all GPUs with the approximated equal access speed, which is vital to facilitate a more even distribution of remote workloads to GPUs in terms of their size and unit access costs. In addition, NEs are generally large in terms of size (due to high dimensionality), which are beyond the device memory limit of a single GPU. Therefore, NEs are ideal to be placed in shared global memory space with sufficient space (with aggregated memory of different GPUs), which also provides direct remote access support across GPUs. Specifically, we will partition the NEs of input graphs into *n* equal-sized partitions (where *n* is the number of GPUs) and place each of them in one GPU’s shared global memory space.

Secondly, for the efficiency of individual pipelines, we allocate the “private” global memory space for storing partitioned graph structure (GP) data, which is only visible to kernels on the current GPU. Our key insight is that GP (e.g., edge lists), is all scalar values and usually small in size, and will only



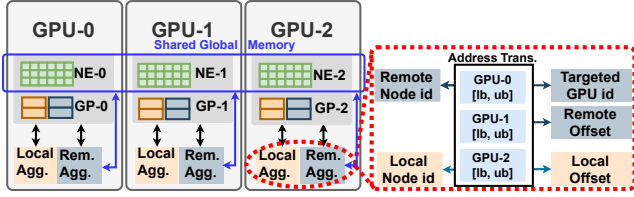


Figure 4: MGG Storage Layout and Communication Pattern. Note that “NE-*i*” is the node embedding partition stored on the *i*-th GPU. “GP-*i*” is the neighbor partition processed by the *i*-th GPU. “GPU-*i* [lb, ub]” is the node-id range [lowerbound, upperbound] of the node embeddings on the *i*-th GPU.

be accessed by the local GPU. Therefore, GP is ideal to be placed in individual GPUs’ DRAM. Such a placement is also important to reduce unnecessary and inefficient remote access on those tiny scalars for fewer pipeline bubbles. In our design, GP data (e.g., edges) from private GPU global memory will be processed by a *address translation* unit for fetching correct NEs on local/remote GPU since the NE indices are rebased to zero on each GPU (Figure 4 right).

## 5 GPU-aware Pipeline Mapping

Efficient pipelining also demands effective mapping of well-constructed pipeline workload and their schedules to the low-level GPU logical processing units (e.g., GPU threads/warp-s/blocks) to overlap computation and communication. To achieve this, we propose *Warp-based Mapping & Pipelining* and *Specialized Memory Design & Optimization* to jointly optimize the pipeline execution efficiency, GPU utilization, and end-to-end design flexibility.

### 5.1 Warp-based Mapping & Pipelining

An effective pipeline mapping demands comprehensive consideration of two major aspects. 1) *Which type of GPU logical processing units (e.g., warps, blocks) should be used for pipeline workload partitions?* We choose GPU warp as the basic working unit to handle the workload of each partition. This is because threads in a warp can collaboratively work on different dimensions of a node embedding simultaneously. Whereas using a single or several threads (less than the size of a warp, 32 threads) would hardly explore the computation parallelism and would cause warp-level divergence. Besides, NVSHMEM remote access initiated by a warp of threads would merge the requests into one remote memory transaction to amortize the overhead. 2) *Which pattern of mapping should be used for benefiting pipeline execution efficiency?* The most straightforward way is to continuously map the neighbor partitions from the local and remote workload list to GPU warps with continuous IDs (Figure 5). However, this strategy would easily suffer from workload imbalance among GPU SMs. This is because warps with continuous IDs are

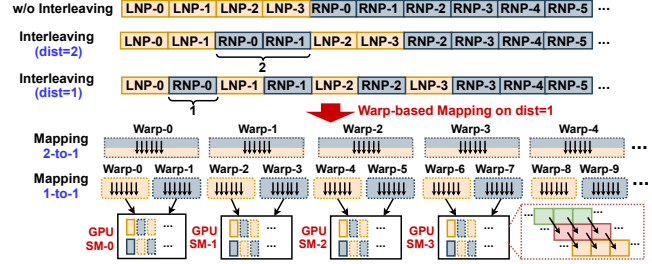


Figure 5: Warp-based Mapping and Pipelining. Note that “LNP” refers to the local neighbor partitions; “RNP” refers to the remote neighbor partitions. Workload and Warps are matched based on colors. Tiny boxes in GPU SM indicate decomposed workload operations for overlapped execution.

more likely to be placed into the same thread block, which is assigned to one SM for LNP processing. Therefore, SMs assigned with warps for handling remote neighbor partitions would lead to much longer latency than SMs assigned with warps for processing local neighbor partitions. Such a workload imbalance would lead to poor GPU utilization and runtime execution performance.

To this end, we introduce our novel *workload interleaving* strategy to balance the workload among SMs on GPUs. Each warp of threads running on GPU would handle one or more pairs of local/remote workload partitions. To more precisely calibrate the warp-to-SM mapping for different pipeline stages to achieve efficient pipelining, we introduce a new metric – *interleaving distance*. We give examples with the interleaving distance equals 1 and 2 for illustration (Figure 5). By mixing different types (both local and remote) of workload together, better GPU utilization can be achieved since when one warp is blocked for high-cost remote access, other warps that are working on local computation can still be served by the SMs warp scheduler for filling up these idle GPU cycles. Moreover, such a design would improve design flexibility. For instance, given an input graph with a selected neighbor partition size, we can adjust the size of interleaving distance and the workload per warp so that waiting cycles of the remote access can be hidden by the computation cycles of the neighbor aggregation. Thus, each warp can be fully utilized while the design can achieve sufficient parallelism.

MGG currently processes the neighbors of adjacent nodes (based on node-ids) to the same thread block where the same block will be scheduled on the same SM. If there are common remote neighbors for those adjacent nodes, their remote requests will be merged. Improving such locality requires reordering the graph nodes to maximize their common neighbors. Such an exploration is orthogonal to our current contribution. In future GPUs, there is a trend to explore the locality among independent processing units. For instance, in Hopper, several thread blocks can be grouped together as thread-block groups. We can explore the tradeoff between the locality benefits and group synchronization overhead.



## 5.2 Specialized Memory Design & Optim.

Efficient software pipelining also demands careful management of high-bandwidth shared memory for promoting data access efficiency and asynchronous primitives for exploiting intra-warp operation pipelining.

**GPU SM Shared Memory Layout:** Based on our MGG’s warp-based workload design, we propose a *block-level shared memory orchestration* to maximize the performance gains. We have several key insights for such a dedicated memory layout design within each thread block. *First*, our neighbor-partition-based workload will generate the intermediate results that can be cached at the high-speed shared memory for reducing the frequent low-speed global memory access. *Second*, NVSHMEM-based remote data access demands a local scratch-pad memory (e.g., registers, shared and global memory) to hold the remote data for local operations.

For the *local* neighbor aggregation, we reserve a shared memory space with  $D$  ( $D$  is the embedding dimension) floating-point numbers for embeddings of the target node in each neighbor partition so that threads from a warp can cache the intermediate results of partial reduction in shared memory. For the *remote* neighbor aggregation, the shared memory space is doubled  $2 \times wpb \times D$  ( $wpb$  is the warps per block). The reason is that we need the first half  $wpb \times D$  for caching the partial aggregation results of each warp and the remaining for the remotely accessed neighbor embeddings. For each MGG kernel design, we will first identify the warp-level information, like warp IDs. Then within each thread block, we define the customized shared memory layout by splitting the contiguous shared memory address into three different parts for neighbor ids, partial aggregation results, and the remotely-fetched node embeddings. We use the dynamic shared memory for design flexibility since those parameters (e.g.,  $wpb$  and  $D$ ) can only be determined at runtime. During execution, we will first calculate the total shared memory size per block and then pass it as a kernel launching parameter.

**Pipelined Memory Operation:** §5.1 has discussed assigning local (LNP) and remote (RNP) neighbor aggregation workloads to warps so that different warps can overlap their computation and communication to fully saturate the active cycles of the GPU SM scheduler. However, only exploiting the inter-warp communication-computation overlap is not enough to maximize the utilization of GPU resources. We further explore the overlapping of the computation and communication at the intra-warp level by carefully scheduling the memory operations. Figure 6(a) shows the case with two LNPs and two RNPs by using the synchronized remote access, we can just sequentially process the two LNPs and the two RNPs. The long-latency remote access can happen only after the completion of its preceding LNP. This could lead to a longer GPU stall for memory operations and low GPU SM utilization. Our profiling also shows that without overlapping, the remote access usually dominates the overall execution

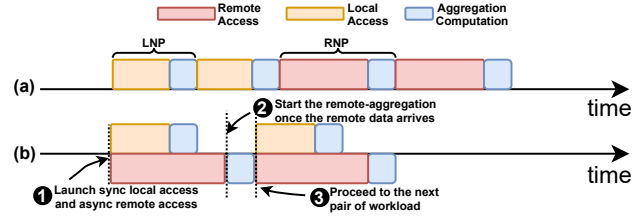


Figure 6: Illustration of (a) w/o and (b) w/ asynchronous primitives for overlapping computation and communication of an individual warp. Note that the length of each rectangular box indicates the estimated latency cost of each operation.

(around 60% of overall latency) compared to the time for local data access plus the time for aggregation computation (around 40% of overall latency). Such observation justifies our design to mainly hide the latency from remote access.

To amortize the cost of remote access for each warp, we introduce *asynchronous remote memory operations* (Figure 6(b)). This improved design consists of two major steps. First, we can simultaneously launch the local memory access while initializing the remote memory access for fetching the node embedding (❶), therefore, the time for remote access can be amortized by the processing of LNP. Second, once the remote access is completed, the current warp will start aggregation on the remotely-fetched node embedding data (❷). The next step will start the new iteration of the previous two steps, which will process a new pair of LNP and RNP.

## 6 Intelligent Runtime Design

In this section, we will discuss our intelligent runtime design with performance/resource analytical modeling and heuristic-based cross-iteration optimization strategy.

**Performance-Resource Analytical Modeling:** The performance/resource model of MGG has two variables: *workload per warp (WPW)* and *shared memory usage per block (SMEM)*, which can be measured by

$$\begin{aligned} WPW &= 2 \cdot ps \cdot D \cdot dist, \\ SMEM &= ps \cdot wpb \cdot IntS + 2 \cdot wpb \cdot D \cdot FloatS \end{aligned} \quad (1)$$

where  $ps$ ,  $wpb$ , and  $D$  are the sizes of neighbor partition, warp per block, and node embedding dimension, respectively;  $dist$  is the interleaved distance of local/remote workloads (§5.1);  $IntS$  and  $FloatS$  are both 4 bytes on GPUs. To determine the value of the  $ps$ ,  $wpb$ , and  $dist$  of a given input graph, we will first compute the total number of warps by using

$$numWarps = \frac{\max\{local, remote\}}{dist} \quad (2)$$

where *local* and *remote* are the number of local and remote partitions, respectively. Then we compute the total number of

blocks and the estimated block per SMs by using

$$\begin{aligned} numBlocks &= \frac{numWarps}{wpb}, \\ blocksPerSM &= \frac{numBlocks}{numSMs} \end{aligned} \quad (3)$$

Later, based on our micro-benchmarking results on diverse datasets, we define our parameter search space and constraints: 1)  $ps \in [1 \dots 32]$  to balance the computation parallelism and synchronization overhead; 2)  $dist \in [1 \dots 16]$  to effectively overlap the computation and remote memory access; 3)  $wpb \in [1 \dots 16]$  to maintain SM warp scheduling flexibility for better occupancy and throughput; 4)  $numSMs \leq c_1$ ,  $SMEM \leq c_2$ , where  $c_1$  and  $c_2$  are hardware constraints [48], e.g., NVIDIA A100 has 108 SMs and 164KB shared memory per SM.

**Heuristic-based Cross Iteration Optimization** To optimize the design of MGG, the parameter  $ps$ ,  $dist$ , and  $wpb$  are initialized as the value 1 at the beginning. Then we optimize one parameter in each of the following iterations. *First*, we increase the  $ps$  to maximize the warp utilization. When further increasing the  $ps$  would also increase the latency, we would stop the search on  $ps$  and switch to  $dist$ . *Second*, we apply a similar strategy to locate the value of  $dist$  that can maximize the overlap of local computation and remote access. *Third*, we increase  $wpb$  to maximize the utilization of the entire SM. If any increase of  $wpb$  would increase the latency, we know that there may be too large thread blocks or too heavy workloads on individual warps that lower SM warp scheduling efficiency or computation parallelism. We would “retreat” (i.e., decrease)  $ps$  to its second-highest value if necessary and restart the increase of  $wpb$ . This optimization algorithm will stop when any decrease of  $ps$  and increase of  $wpb$  would lead to higher latency than the top-3 lowest latency. The latency of each iteration during the optimization will be recorded by a configuration lookup table. Finally, the configuration with the lowest latency will be applied.

This particular optimization order of parameters ( $ps$ ,  $dist$ , and  $wpb$ ) is based on two major aspects: (i) *Spatially speaking*, the granularity is from coarse-grained algorithm-level partitioning through  $ps$ , to medium-grained pipeline construction through  $dist$  (according to the partition plan), to fine-grained pipeline-to-warp fine-tuning through  $wpb$  (according to the pipeline design). (ii) *Temporally speaking*, the three optimizations are applied at loading-time ( $ps$  to decide layout), kernel initialization ( $dist$  to decide pipeline), and runtime ( $wpb$  to decide pipeline mapping), respectively.

The above parameter adaption for dynamic pipelining is vital for design/optimization generality. This is because the characteristics of graphs (#nodes/edges and embedding sizes) would lead to different efficiency of kernel pipelines. Our later experimental studies (as shown in Figure 11) demonstrate its benefits with up to 70% of performance improvements.

Table 1: Datasets for Evaluation.

Dataset	#Vertex	#Edge	#Dim	#Class
reddit(RDD) [45]	232,965	114,615,892	602	41
enwiki-2013(ENWIKI) [23]	4,203,323	202,623,226	300	12
it-2004(IT04) [10]	41,291,594	1,150,725,437	256	64
ogbn-paper100M(PAPER) [12]	111,059,956	1,615,685,872	128	64
ogbn-products(PROD) [17]	2,449,029	61,859,140	100	47
ogbn-proteins(PROT) [17]	132,534	39,561,252	8	112
com-orkut(ORKT) [23]	3,072,441	117,185,083	128	32

## 7 Evaluation

**Benchmarks & Datasets** Despite the diversity of GNN models, the fundamental computation and communication paradigm (vector-based scatter-gather operation) in multi-GPU GNNs remains the same. We evaluate two distinctive and representative GNN models on *node classification* tasks:

The first type of GNN model uses a *non-discriminated* neighbor aggregation strategy, where all neighbors contribute equally when doing the aggregation. We choose **Graph Convolutional Network (GCN)** [21], which is the most popular GNN model and is also the key backbone network for many other GNNs, such as GraphSAGE [16] and Differentiable Pooling [52]. We use *2 layers with 16 hidden dimensions* for GCN, which is also the setting from the original paper [21]. The computation of a 2-layer GCN can be expressed as

$$Z = \text{Softmax}(\hat{A} \text{ReLU}(\hat{A}XW^1)W^2). \quad (4)$$

where  $\hat{A}$  is the adjacent matrix of the input graph with self-loop edges, and  $X$  is the input node embedding matrix, where  $X \in R^{N \times D}$ ;  $N$  is the number of nodes in a graph;  $D$  is the size of node embedding dimensions.  $W^1$  and  $W^2$  are trainable weight matrices in layer-1 and layer-2, respectively.

The second type uses a *discriminated* neighbor aggregation strategy, where neighbors would contribute differently depending on their calculated edge-specific features. We choose **Graph Isomorphism Network (GIN)** [49], which aims to distinguish the graph structure that cannot be identified by GCN. Each layer of GIN can be expressed as

$$h_v^{l+1} = MLP^l((1 + \varepsilon^l)h^l + \sum_{u \in N_{(v)}} h_u^l). \quad (5)$$

where  $l$  is the layer ID and  $l \in \{0, 1\}$ ,  $MLP$  is a fully-connected neural network,  $h_v$  is the node embedding for node  $v$ , and  $N_{(v)}$  stands for the neighbors of node  $v$ . GIN mainly differs from GCN in its aggregation function, which introduces a weight parameter as the ratio of contribution from its neighbors and the node itself. In addition, GIN is the reference architecture for many other advanced GNNs with more edge properties, such as Graph Attention Network [43]. For GIN evaluation, we use *5 layers with 64 hidden dimensions*, which is also the setting used in the original paper [49]. Graphs (Table 1) used in our evaluation are large in their number of nodes and edges that demand multi-GPU capability for effective GNN computation. #Class is the output dimension

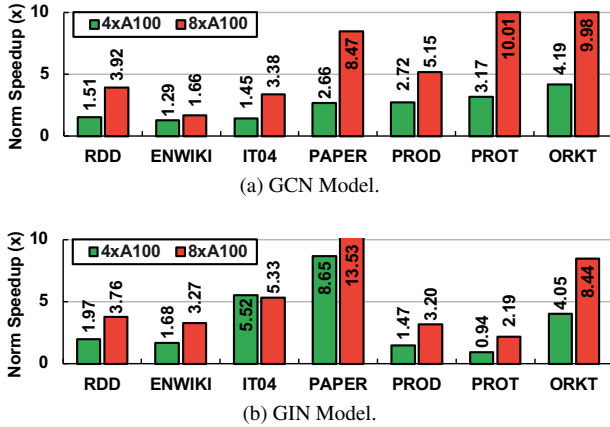


Figure 7: Performance comparison with DGL. Note that full-graph PAPER on DGL requires A100-80GB.

(#labels) for the node classification task. #Dim is the embedding dimension of the input graph.

**Baselines** In this evaluation, we compared MGG with several existing systems that support large full-graph GNN (i.e., caching the entire graph on GPUs) on multi-GPU platforms.

**1) Deep Graph Library (DGL)** [45] is the state-of-the-art framework for large-scale GNNs across GPUs. It leverages PyTorch-Direct [27] as the communication backend for GPU-initiated zero-copy memory access [41] to fetch neighbors embedding from the CPU host. **2) MGG-UVM** [20] is a GNN design by adapting MGG to leverage unified virtual memory (UVM). UVM has been highlighted in handling irregular graph computations (such as PageRank) on large graphs [20]. However, [20] is not open-sourced, we thus generalize the pipeline kernel designs and optimizations (§4 and §5) of MGG to build such a UVM baseline and incorporate optimizations from [20]. Note that UVM and zero-copy memory are different communication backends [1]. Thus, MGG-UVM does not implement zero-copy data transfer. We remark UVM is the key communication protocol before the new hardware support for fine-grained direct GPU-GPU communication (e.g., NVSHMEM). UVM is more coarse-grained and will require the engagement of CPUs (e.g., host memory management) for communication. The reason to use MGG-UVM is to show that if there is no advanced hardware support (e.g., NVSHMEM) for fine-grained direct GPU-GPU communication, the benefits of our elaborated pipeline can be offset by UVM communication overhead. **3) ROC** [18] is a popular distributed multi-GPU system for full-graph computation. ROC highlights its learning-based partitioning and leverages NVIDIA Legion [5] runtime for communication and task scheduling.

Other multi-GPU GNN designs, like NeuGraph [26] and P3 [12], are not publicly available. Initially, we plan to evaluate MGG on AMD ROC\_SHMEM [2]. However, as indicated in its document, the existing ROC\_SHMEM is an experimental prototype and is not officially ready to be applied in prac-

tice due to very strict software limitations (e.g., only supports ROCm v4.3) and hardware (e.g., only supports AMD GFX9 GPUs), which are quite challenging to find and deploy and not supported by any existing GNNs frameworks [6, 18, 28] for comparison. We believe that once ROC\_SHMEM becomes ready and generally applicable, MGG can be easily migrated to AMD multi-GPU platforms.

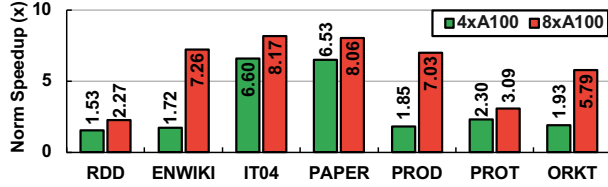
There is no existing design that can leverage GPU-to-GPU communication only for distributed full-graph GNN computation. We try our best to measure the best-possible baseline performance. DGL and ROC have longer latency in the earlier iteration due to cache warmup for node embedding on GPU memory. We thus perform warm up iterations until their per-iteration latency becomes stable, and then measure their performance with minimized CPU-GPU data movements.

**Platforms & Tools** The implementation of MGG consists of ~9K LoC. We compile and link MGG with CUDA (v11.2), OpenMPI (v4.1.1), NVSHMEM (v2.0.3), and cuDNN (v8.2) library. Our major platform is an NVIDIA DGX-A100 with dual AMD Rome 7742 processors (each with 64 cores, 2.25 GHz), 1TB host memory, and 8×A100 GPUs (40 GB) connected via NVSwitch, which offers 600 GB/s GPU-to-GPU bi-directional bandwidth. For the modeling study, we also leverage DGX-1 with 4×V100 GPUs connected via NVLinks. We use NVIDIA NSight Compute to get the kernel-level profiling metrics. Speedup is averaged over 100 runs.

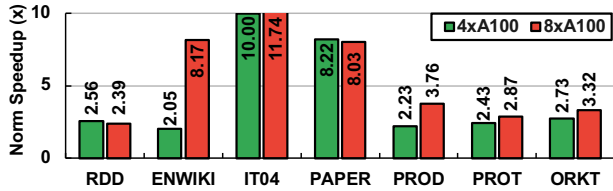
## 7.1 End-to-End Performance

**Compared with DGL** In this section, we will compare with the state-of-the-art DGL framework, which leverages PyTorch-Direct for cross-GPU communication. We evaluate different datasets and platform settings (with 4 and 8 A100 GPUs). As shown in Figure 7, MGG outperforms DGL with averaged  $4.25\times$  and  $4.57\times$  speedups on GCN and GIN models, respectively. We also notice a trend that MGG demonstrates a more pronounced speedup with more GPUs. With the increasing number of GPUs, DGL suffers from heavy memory access contention, since multiple GPUs are initiating massive requests to access the neighbor embeddings on the CPU host memory. Another observation is that on GIN ( $D = 64$ ) with higher hidden dimensionality for smaller datasets (e.g., PROD and PROT), the performance gap between DGL and MGG is smaller compared to GCN ( $D = 16$ ) since as indicated in [28], zero-copy memory would be beneficial from more coarse-grained data movement (with larger embedding vector) that can saturate the PCIe cache line (128 Bytes). While such an advantage of DGL diminishes for those larger datasets (e.g., IT04 and PAPER) on GIN due to significantly increased sparsity and irregularity. In addition, compared with MGG, DGL assumes the one-size-fits-all communication strategy would work well for all input datasets. Therefore, it ignores the importance of the inputs and hardware properties, which would bring non-trivial (more than 30%) benefits (§7.2).





(a) GCN Model.



(b) GIN Model.

Figure 8: Performance comparison with MGG-UVM.

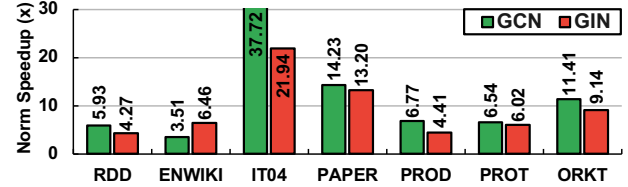
Table 2: Additional performance comparison of MGG and DGL on GraphSAGE and GAT.

Model	RDD	ENWIKI	IT04	PAPER	PROD	PROT	ORKT
SAGE	4.97×	1.76×	1.99×	3.53×	7.05×	3.39×	3.53×
GAT	2.65×	1.62×	2.06×	3.04×	2.06×	3.39×	3.04×

MGG can also be extended to cover other GNN models. The following results show the speedups of MGG over DGL on GraphSAGE with layerwise node neighbor sampling and GAT with dot-product edge attention. Table 2 shows that the performance results of GAT and SAGE also agree with our prior observations on the GCN and GIN, demonstrating the generality and effectiveness of our proposed design and optimizations to handle more complex dataflow (e.g., edge attention and softmax) in multi-GPU GNN computation.

Despite that MGG (NVSHMEM) and DGL (with CPU-GPU zero-copy memory [41]) both rely on GPU-initiated communication and overlap communication with computation, their underlying mechanism is different, and MGG shows more performance advantages. MGG can leverage inter-GPU communication while DGL can only rely on CPU-GPU communication with limited bandwidth. This makes the communication costs pronounced in DGL and offsets the performance gains from massive thread-level parallelism. This experiment also shows that MGG can serve as a drop-in replacement for the existing communication backend of DGL to improve large-scale full-graph GNN computation.

**Compared with MGG-UVM** In this experiment, we compare MGG with its UVM-based counterpart, MGG-UVM, which uses UVM in place of NVSHMEM for remote communication. Figure 8 shows that MGG achieves  $4.58\times$  speedup and  $5.04\times$  speedup on average compared to MGG-UVM on GCN and GIN, respectively. The MGG-UVM leverages the page-faulting-based remote data access that is more coarse-grained (around 4 KB) in comparison with a single node embedding size (less than 0.4KB), which leads to higher overhead and lower effective bandwidth usage per embedding

Figure 9: Performance comparison with ROC with  $8\times$ A100.

transfer. Such an overhead would exacerbate with more GPUs and also make MGG-UVM challenging for GPU SM schedulers to effectively dispatch instructions for the next available warps. This is mainly because most of the warps wait for the long-cycle page-faulting and migration.

We notice that with the increase of the dimension size (i.e., data movement granularity), the speedup over MGG-UVM becomes higher. We later found out that the increase of data-movement granularity actually increases the overall page-fault counts. This is because embedding vectors are generally stored continuously for memory efficiency instead of aligning with the size of memory pages. Therefore, increasing the size of individual embedding also increases the likelihood of triggering multiple pagefaults per embedding transfer.

Comparing among datasets, for graphs (e.g., PAPER) with more nodes/edges and lower average node degree, MGG would demonstrate more speedups since these graphs exhibit more irregular and sparse access that can not well fit into regular fix-sized pages. This also indicates the importance of amortizing communication overhead. Thanks to pipeline-centric workload management, we can effectively amortize such costs with careful operation scheduling.

We further measure two performance-critical GPU kernel metrics that are the key indicators of our pipeline efficiency (§4.1): *Achieved Occupancy* (the ratio of the average active warps per active cycle to the maximum number of warps supported in an SM) and *SM utilization* (the utilization of all available SMs on a single GPU). MGG improves SM utilization (by 21.15% on average) and occupancy (by 39.20% on average) compared to MGG-UVM. This indicates that MGG can effectively 1) distribute irregular workloads to SMs to balance workloads among pipelines and improve the overall GPU utilization, and 2) overlap the remote access and local aggregation computation from different warps to reduce pipeline bubbles and maximize SM occupancy.

**Compared with ROC** In this experiment, we compare MGG with ROC [18] on their officially released GCN model implementation. We originally plan to evaluate both 4 and 8 GPU settings. However, ROC reports many out-of-memory (OOM) errors for those large graphs on GCN/GIN model and medium graphs on the GIN model due to its aggressive caching of those intermediate tensors on GPUs. Therefore, we keep our comparison to 8 GPUs. Performance-critical ROC runtime configurations (e.g., #CPU cores, GPU/host memory size) are optimized to fully utilize the DGX-A100.



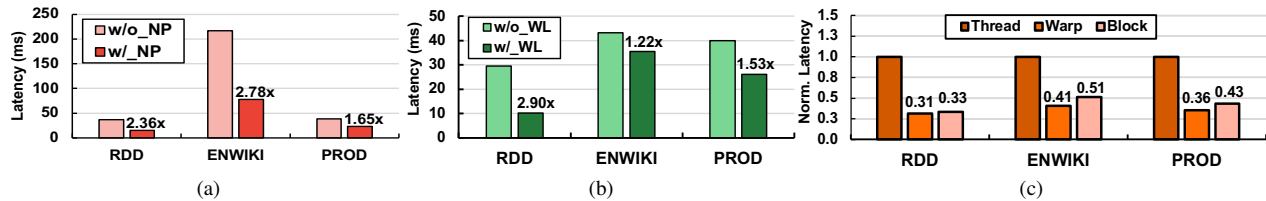


Figure 10: Optimization Analysis: (a) Neighbor Partitioning; (b) Workload Interleaving; (c) Choice of Communication Primitives.

Figure 9 shows that MGG achieves averaged  $12.30\times$  and  $9.35\times$  speedups over ROC on GCN and GIN, respectively. MGG demonstrates a more pronounced speedup over ROC on the larger graph (e.g., IT04 and PAPER), which has more irregular neighbor embedding access. The Legion runtime of ROC relies on the DMA engine for bulky data (batched embeddings) transfer between host and GPU memory, leading to higher throughput but inferior latency performance. Besides, ROC relies on a separate communication-computation design, where computation happens after the full completion of communication. Such a design eliminates the opportunity to fill idle GPU cycles with computation during communication. In addition, the learning-based partitioning (to reduce communication) of ROC shows benefits on relatively smaller datasets (e.g., RDD and PROT) but hard to find optimal partition plans for large graphs due to the input structure complexity.

## 7.2 Optimization Analysis

**Neighbor Partitioning (NP)** We compare MGG with a baseline design without applying the neighbor partitioning technique (*i.e.*, each aggregation workload consists of all local/remote neighbors) on  $4\times A100$ . We apply the workload interleaving for both implementations and fix the warp-per-block size to 2 to eliminate the impact from other performance-related factors. Figure 10(a) shows higher latency (averaged  $2.26\times$ ) for designs without applying neighbor partitioning, since the workload imbalance becomes more severe across different warps without neighbor partitioning, especially for those graphs with many remote access demands, leading to limited computing parallelism and GPU underutilization.

**Workload Interleaving (WL)** We compare MGG with a baseline design without workload interleaving (*i.e.*, remote neighbor aggregation and local neighbor aggregation are mapped separately to the GPU warps). We fix the neighbor partition size to 16 and the warp-per-block size to 2. Figure 10(b) shows that MGG consistently outperforms the non-interleaved baseline with an average of  $1.89\times$  speedup. Without interleaving the local/remote workload, the workload distribution would be highly skewed, where the heavy and intensive remote aggregation would be gathered on certain warps close to each other while the lightweight local aggregation would be gathered on some other warps close to each other. This leads to inefficient warp scheduling and higher latency.

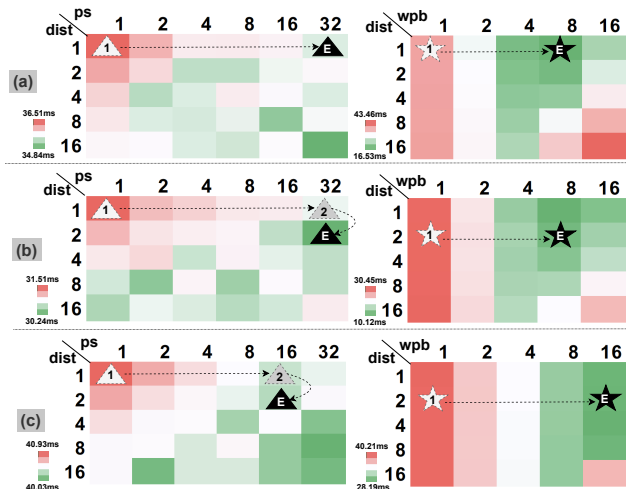


Figure 11: Parameter selection for three different settings. (a), (b), and (c) are for setting I, II, and III, respectively. Note that the left-side figures show the runtime latency for different combinations of  $ps$  and  $dist$ , while the right-side figures show the latency for different combinations of  $wpb$  and  $dist$ . The solid black triangle with “E” is the searched “optimal” combination for  $ps$  and  $dist$ , while the black solid star with “E” is the searched “optimal”  $wpb$  given  $dist$  and  $ps$ .

**Communication Primitives** We adopt MGG with different NVSHMEM primitives at the *thread*, *warp*, and *block* levels. We fix the number of GPUs to 2, the hidden dimension to 16, the neighbor partition size to 2, and the distance of workload interleaving to 2. Figure 10(c) shows that warp-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_warp_get`) for remote accessing can bring the lowest latency. For thread-level NVSHMEM primitives (*e.g.*, `nvshmem_float_get`), it would not coalesce the remote memory access to reduce unnecessary transactions. For the block-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_block_get`), the higher overhead comes from collaborating a block of threads for remote access, since thread blocks (usually consisting of multiple warps) is larger than a single warp, thus, leading to higher synchronization and scheduling cost. This study also shows that our choice of warp-level primitives strikes a good balance between memory access efficiency and scheduling flexibility.

**Modeling and Optimization** We further analyze the effectiveness of our lightweight analytical model for design space

Table 3: Accuracy-Latency of GNNs w/ and w/o sampling.

Dataset	Accuracy w/ sampling	Accuracy w/o sampling	Latency (w/o vs. w/ sampling)
RDD	0.937	0.957	1.07×
PROT	0.776	0.825	1.25×

search. Specifically, three key parameters are studied, the size of neighbor partitioning ( $ps$ ), the interleaving distance ( $dist$ ), and the warps per block ( $wpb$ ). We consider three different settings on a 2-layer GCN model: I: RDD on  $4 \times A100$  as the basic setting. II: RDD on  $8 \times A100$  to demonstrate the adaptability toward the different numbers of GPUs. III: RDD on  $4 \times V100$  [37] to demonstrate the adaptability toward the different types of GPUs. We decompose searching results into two parts corresponding to the output of the *second* and *third* steps of the optimization discussed in §6.

Figure 11 shows that our performance modeling and parameter selection strategy can pinpoint the low-latency design for the above three settings. The overall searching process only requires about 10 iterations to reach the final “optimal” settings. Note that here we show latency results for all possible settings for comparison. While in practice, we only need to traverse a small part of the whole design space (as indicated by the boxes touched by the dot lines). By comparing the final optimal runtime configuration setting and the initial configuration, we can see that modeling and cross-iteration optimization can decrease the execution time by up to 68%. In the end-to-end GNN training (usually more than 100 iterations), such a latency saving would also be significant.

### 7.3 Additional Study

**Accuracy-latency Tradeoff** This study will analyze the accuracy-latency tradeoff between GNNs with sampling and full-graph (w/o sampling) on  $8 \times A100$ . Table 3 shows an evident node classification accuracy increase (2% to 5%) of GNN w/o sampling over GNN w/ sampling. The accuracy of sampling-based GNN would be affected by many factors (e.g., sampling rate at each GNN layer and graph structure). It is thus highly tricky to choose the “optimal” value for those factors. Here we follow the conventional way for GNN sampling [45]. The accuracy difference agrees with previous GNN algorithmic work [16]. In many real-world applications (e.g, e-commerce), such an accuracy advantage of full-graph GNNs are be more preferred by users. Because even 1% accuracy would make significant profit gains when deploying services at scale while the latency penalty is relatively minor.

**Generality to other applications** The design of MGG can be generalized to other similar applications. We demonstrate the typical and popular deep-learning recommendation model (DLRM) [31, 47, 54] that has been widely used in the industry. In multi-GPU DLRM, the large embedding tables are partitioned by rows and stored in different GPUs. The DLRM inputs (embedding access queries) will request embeddings

Table 4: DLRM [31] with MGG in Embedding Lookup.

Implementation	DLRM [31]	DLRM (MGG)
Time (ms)	315.27	119.66

from tables on different GPUs and then apply operations (e.g., elementwise addition or dot product) on those fetched embeddings. Such embedding lookup is highly sparse and irregular and dominates ( $> 80\%$  latency [15, 54]) the overall DLRM computation. We improve the mainstream DLRM system [31] with the design and optimizations of MGG to accelerate embedding lookup and element-wise addition and compare with the original system (which relies on NCCL) [31] under 4-GPU settings on the popular Criteo Kaggle [9] dataset. Table 4 shows that DLRM with MGG effectively reduces the lookup time (2.64 $\times$ ). The fine-grained remote access of MGG can reduce redundant inter-GPU traffic by using NCCL and offset the cost by massively parallel GPU-initiated communication.

## 8 Discussion

**Deep Learning Pipelines:** Despite the popularity of the pipeline concept in the conventional dense DL, the generalization of such a technique in sparse GNN computation is yet to be explored in-depth. PiPAD [44] overlaps the communication (CPU-to-GPU) and processing (on GPUs) between adjacent graph partitions. Adopting this strategy, we will get designs as Figure 3(c), which would still suffer from pipeline bubbles due to workload imbalance. vPipe [56] dynamically assigns a DNN layer to certain pipeline stages during the runtime. It improves pipeline efficiency and GPU utilization for DNN models. However, adopting this approach in our fine-grained kernel pipeline would incur high overhead due to frequent workload reassignment and context switching. In addition, the pipeline bubbles in dense DNN are predictable, input-agnostic, and can be reduced offline. However, the pipeline bubble for GNN can only be figured out at runtime due to input dependency. It, therefore, demands careful online workload balance and a pipeline schedule/mapping.

**Graph Partitioning Strategies:** Besides our current ID-based graph partitioning, our designs/optimizations could also be extended to support other graph partitioning strategies from prior graph processing and GNN work. There are several major categories. 1) *Locality-driven partitioning* (e.g., Gemini [57] and Rabbit order [4]) minimizes the communication/synchronization cost in distributed graph processing/GNN computing. Such partition strategies are orthogonal to our current design optimization. Despite it will reduce the total size of communication, the communication pattern remains the same with irregular, sparse, and fine-grained data movements. Our MGG design can be modified to accommodate such reduced-communication cases through dynamic kernel re-configuration (e.g., fine-tuning the interleaving distance and warp-to-block mapping) to maximize

communication and computation efficiency. 2) *Workload-driven partitioning* (e.g., NeuGraph [26] and CUBE [53]) balances the irregular graph/GNN workload among different devices. This type of strategy typically maintains multiple replicas of nodes and node properties on different devices and synchronizes partial results in replicas after local computation on each device. Our current design be adapted to handle such cases by inserting device synchronization primitives (NVSHMEM collective communication primitives, such as `nvshmem_float_sum_reduce`) for maintaining data consistency among different replicas. 3) *Learning-based partitioning* (e.g., ROC [18]) dynamically learns an “optimal” partitioning strategy that can maximize the computation performance. Our current design/optimization can also support this partitioning strategy by incorporating the overhead of NVSHMEM remote memory access in the runtime prediction model when optimizing partitioning strategies online.

## 9 Conclusion

This paper presents MGG, a novel multi-GPU system design, and implementation to exploit the potential of leveraging GPU intra-kernel software pipeline for accelerating GNNs. MGG consists of GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping, and an intelligent runtime design to dynamically improve the GNN runtime performance. Experiments show the advantages of MGG over state-of-the-art solutions and its generality towards other DL applications.

## 10 Acknowledgment

We would like to appreciate the great help and support from OSDI shepherd and anonymous reviewers. This work was supported in part by NSF-2124039 and CloudBank [32]. We also appreciate the generous help and support from Amazon Faculty Research Award 2021 for Professor Yufei Ding and NVIDIA Graduate Fellowship 2022-2023 for Yuke Wang.

## References

- [1] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, 2013.
- [2] AMD. Rocm openshmem. [https://github.com/ROCm-Developer-Tools/ROC\\_SHMEM](https://github.com/ROCm-Developer-Tools/ROC_SHMEM).
- [3] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.
- [4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.
- [5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.
- [6] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.
- [7] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*, 2005.
- [8] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations (ICLR)*, 2018.
- [9] Criteo. Criteo display ad challenge. <https://kaggle.com/c/criteodisplay-ad-challenge>.
- [10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 2011.
- [11] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS)*, 2017.
- [12] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [13] Jaume Gibert, Ernest Valveny, and Horst Bunke. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition*, 2012.
- [14] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.
- [15] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagan, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia,

- Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook’s dnn-based personalized recommendation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.
- [16] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS)*, 2017.
- [17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems (NeurIPS)*, 33, 2020.
- [18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of the 3rd MLSys Conference*, 2020.
- [19] Riesen Kaspar and Bunke Horst. *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.
- [20] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [21] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.
- [22] Jérôme Kunegis and Andreas Lommatzsch. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.
- [23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <https://snap.stanford.edu/data>, 2014.
- [24] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpubirect. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2019.
- [25] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.
- [26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference (ATC)*, 2019.
- [27] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv preprint arXiv:2101.07956*, 2021.
- [28] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 2021.
- [29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning (ICML)*, 2021.
- [31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Iliia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.
- [32] Michael Norman, Vince Kellen, Shava Smallen, et al. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing*. 2021.
- [33] Nvidia. Dgx superpod. <https://nvidia.com/en-us/data-center/dgx-superpod/>.
- [34] Nvidia. Nvidia collective communication library (nccl). <https://developer.nvidia.com/nccl>.
- [35] Nvidia. Nvidia dgx a100. <https://nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf>.



- [36] Nvidia. Nvshmem communication library. <https://developer.nvidia.com/nvshmem>.
- [37] Nvidia. Tesla v100. <https://nvidia.com/en-us/data-center/v100/>.
- [38] NVIDIA. Unified memory for cuda beginners. <https://developer.nvidia.com/blog/unified-memory-cuda-beginners/>.
- [39] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.
- [40] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *The 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, 2008.
- [41] Tim Schroeder. Peer-to-peer & unified virtual addressing. [https://developer.download.nvidia.com/CUDA/training/cuda\\_webinars\\_GPUDirect\\_uva.pdf](https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf).
- [42] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.
- [43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.
- [44] Chunyang Wang, Desen Sun, and Yuebin Bai. Pipad: Pipelined and parallel dynamic gnn training on gpus. *28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023.
- [45] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [46] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [47] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. El-rec: efficient large-scale recommendation model training via tensor-train embedding table. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.
- [48] wikipedia. Nvidia gpu micro-architecture. <https://en.wikipedia.org/wiki/CUDA>.
- [49] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.
- [50] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygen: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.
- [51] Jianbang Yang, Dahai Tang, Xiaoni Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.
- [52] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *The 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.
- [53] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [54] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 2019.
- [55] Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every document owns its structure: Inductive text classification via graph neural networks. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.
- [56] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, et al. vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2021.
- [57] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

## A Artifact Appendix

MGG is a holistic runtime for exploiting intra-GPU-kernel communication-computation pipelining to accelerate multi-GPU GNNs. MGG consists of two parts. The first part is the host-side CPU program. It is responsible for dataset loading, runtime configuration generation, and invoking the GPU-side program. The second part is the device-side GPU program, called kernels. It is responsible for the major computation and communication of the GNN model on sparse neighbor-aggregation across GPUs and dense node-update phase within each GPU. MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping.

- Code repository: [Github](#)<sup>2</sup> and [Zenodo](#)<sup>3</sup>.
- **Hardware, OS & Compiler:**
  - NVIDIA DGX-A100 with dual AMD Rome 7742 processors (each with 64 cores, 2.25 GHz), 1TB host memory, and 8×A100 GPUs (40 GB) connected via NVSwitch (600 GB/s).
  - Operating systems: Ubuntu 20.04+.
  - Compilers: NVCC (v11.2), GCC (v7.5.0),
  - Libraries: CUDA (v11.2), OpenMPI (v4.1.1), NVSHMEM (v2.0.3), cuDNN (v8.2).
  - Datasets: SNAP [23] and OGB [17].

### Environment Setup

#### Step-1: Download libraries and datasets.

##### – 1.1. Download libraries.

```
wget storage.googleapis.com/mgg_data/local.tar.gz
tar -zxvf local.tar.gz
tar -zxvf local/nvshmem_src_2.0.3-0/build_cu112.tar.gz
```

##### – 1.2. Download datasets and Setup Baselines.

```
wget storage.googleapis.com/mgg_data/dataset.tar.gz
tar -zxvf dataset.tar.gz
cd dgl_pydirect_internal/
wget storage.googleapis.com/mgg_data/graphdata.tar.gz
&& tar -zxvf graphdata.tar.gz
&& rm graphdata.tar.gz
cd ..
gsutil cp -r gs://mgg_data/roc-new/ .
```

##### – 1.3. Launch Docker for MGG.

```
cd docker
./launch.sh
```

##### – 1.4. Compile MGG implementations.

```
mkdir build && cd build && cmake .. && cd ..
./0_mgg_build.sh
```

<sup>2</sup><https://github.com/YukeWang96/MGG-OSDI23-AE.git>

<sup>3</sup><https://doi.org/10.5281/zenodo.7853945>

### Step-2. Run Initial Tests.

Please try below Section-3.4 and Section-3.5.

### Step-3: Experiments.

#### – 3.1. Compare with UVM (Fig.8a and Fig.8b).

```
./0_run_MGG_UVM_4GPU_GCN.sh
./0_run_MGG_UVM_4GPU_GIN.sh
./0_run_MGG_UVM_8GPU_GCN.sh
./0_run_MGG_UVM_8GPU_GIN.sh
```

Results can be found at Fig\_8\_UVM\_MGG\_4GPU\_GCN.csv, Fig\_8\_UVM\_MGG\_4GPU\_GIN.csv, Fig\_8\_UVM\_MGG\_8GPU\_GCN.csv, Fig\_8\_UVM\_MGG\_8GPU\_GIN.csv

#### – 3.2. Compare with DGL (Fig.7a and Fig.7b).

```
cd dgl_pydirect_internal/
./launch_docker.sh
cd gcn/
./0_run_gcn.sh
cd ../gin/
./0_run_gin.sh
```

Results of DGL can be found at 1\_dgl\_gin.csv and 1\_dgl\_gcn.csv. MGG reference is in MGG\_GCN\_8GPU.csv and MGG\_8GPU\_GIN.csv.

#### – 3.3. Compare with ROC on 8xA100 (Fig.9).

```
cd roc-new/docker
./launch.sh
```

Results can be found at Fig\_9\_ROC\_MGG\_8GPU\_GCN.csv, Fig\_9\_ROC\_MGG\_8GPU\_GIN.csv.

#### – 3.4. Compare NP with w/o NP (Fig.10a).

```
python 2_MGG_NP.py
```

Note that the results can be found at MGG\_NP\_study.csv.

#### – 3.5. Compare WL with w/o WL (Fig.10b).

```
python 3_MGG_WL.py
```

Note that the results can be found at MGG\_WL\_study.csv.

#### – 3.6. Compare API (Fig.10c).

```
python 4_MGG_API.py
```

Note that the results can be found at MGG\_API\_study.csv.

#### – 3.7. Design Space Search (Fig.11a).

```
python 5_MGG_DSE_4GPU.py
python 5_MGG_DSE_8GPU.py
```

Results can be found at Reddit\_4xA100\_dist\_ps.csv, Reddit\_4xA100\_dist\_wpb.csv, Reddit\_8xA100\_dist\_ps.csv, Reddit\_8xA100\_dist\_wpb.csv.





# Optimizing Dynamic Neural Networks with Brainstorm

Weihao Cui<sup>1\*</sup>, Zhenhua Han<sup>2</sup>, Lingji Ouyang<sup>3\*</sup>, Yichuan Wang<sup>1</sup>, Ningxin Zheng<sup>2</sup>, Lingxiao Ma<sup>2</sup>, Yuqing Yang<sup>2</sup>, Fan Yang<sup>2</sup>, Jilong Xue<sup>2</sup>, Lili Qiu<sup>2</sup>, Lidong Zhou<sup>2</sup>, Quan Chen<sup>1</sup>, Haisheng Tan<sup>3</sup>, Minyi Guo<sup>1</sup>  
<sup>1</sup>Shanghai Jiao Tong University, <sup>2</sup>Microsoft Research Asia,  
<sup>3</sup>University of Science and Technology of China

## Abstract

Dynamic neural networks (NNs), which can adapt sparsely activated sub-networks to inputs during inference, have shown significant advantages over static ones in terms of accuracy, computational efficiency, and adaptiveness. However, existing deep learning frameworks and compilers mainly focus on optimizing static NNs with deterministic execution, missing optimization opportunities brought by non-uniform distribution of activation in dynamic NNs. The key to optimizing dynamic NNs is the traceability of how data are dynamically dispatched to different paths at inference. Such dynamism often happens at sub-tensor level (e.g., conditional dispatching tokens of a tensor), thus hard for existing tensor-centric frameworks to trace due to misaligned expression granularity.

In this paper, we present Brainstorm, a deep learning framework for optimizing dynamic NNs, which bridges the gap by unifying how dynamism should be expressed. Brainstorm proposes (1) *Cell*, the key data abstraction that lets model developers express the data granularity where dynamism exists, and (2) *Router*, a unified interface to let model developers express how *Cells* should be dynamically dispatched. Brainstorm handles efficient execution of routing actions. This design allows Brainstorm to collect profiles of fine-grained dataflow at the correct granularity. The traceability further opens up a new space of dynamic optimization for dynamic NNs to specialize their execution to the runtime dynamism distribution. Extensive evaluations show Brainstorm brings up to  $11.7\times$  speedup ( $3.29\times$  on average) or leads to 42% less memory consumption for popular dynamic neural networks with the proposed dynamic optimizations.

## 1 Introduction

As deep neural network models become large and complex, it is more and more challenging to sustain the growth of model size due to the increased computing requirement. The key

limitation is the static activation of a whole network regardless of inputs, which is much less efficient than a human brain that can dynamically and sparsely activate neurons related to perceived information. Therefore, there have been numerous efforts by machine learning researchers to design *dynamic* neural networks that can feed inputs into different sub-neural structures or parameters of a large model during inference. Dynamic neural networks have shown favorable properties including efficiency [1–8], adaptiveness [1, 9, 10], generality [1, 9, 11, 12], and interpretability [9, 13]. For example, by designing a large number of expert sub-networks but only conditionally activating a small subset of them, Mixture-of-Expert (MoE) has helped to scale Transformer to trillions of parameters and achieve superior performance [14, 15].

Unfortunately, existing deep learning (DL) frameworks are not yet effective for running dynamic neural networks. Their optimization mainly focuses on static neural networks, whose operator execution order is deterministic for all inputs. It has been widely studied in compilers for general programs (e.g., Java, C#) to leverage runtime characteristics of programs to dynamically optimize their execution [16, 17]. By analyzing runtime profiles of dynamism, we find many dynamic NNs have similar opportunities due to their non-uniform distribution of branch activation or token dispatching, which can be further utilized for dynamic optimization.

However, existing tensor-centric programming models cannot support dynamic optimization well. The major challenge is the misaligned expression granularity, i.e., tensor-centric compilers can only trace how data flows at the tensor level in a static dataflow graph (DFG), but dynamism often happens at the sub-tensor level in dynamic NNs. For example, Mixture-of-Experts (MoE) networks use hidden dimensions within input tensors to represent the concept of “tokens”, which are dynamically reordered to activate parallel expert sub-networks with different tokens. It is critical for dynamic optimizations to collect profiles of dynamism, which is hard for existing compilers because they have no knowledge about what “tokens” are and how they are dynamically dispatched.

In this paper, we present Brainstorm, the first framework to

\*This work is done while Weihao Cui and Lingji Ouyang are interns in Microsoft Research



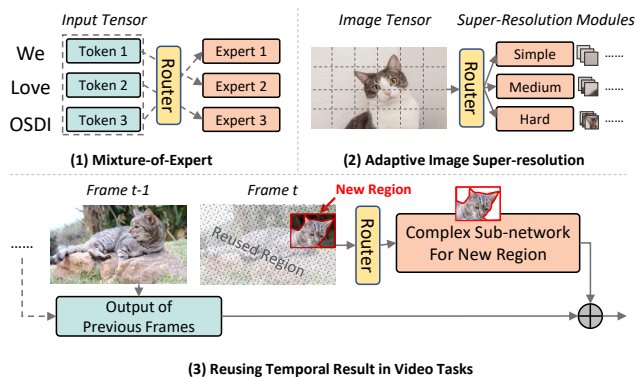


Figure 1: Examples of dynamic neural networks routing at token-level, patch-level, and pixel-level.

optimize the execution of dynamic NNs. Brainstorm unifies the expression of dynamic NNs to make their dynamism easy to trace. At the core of Brainstorm is a new data abstraction called *Cell* that lets model developers describe the granularity of dynamism, e.g., a token inside a tensor. To make *Cell*-level dataflow traceable, Brainstorm unifies the *Router* interface to let model developers express how *Cells* should be dynamically dispatched among multiple branches. Brainstorm can collect the runtime profiles of *Routers* with negligible overhead. Inspired by profile-guided optimization of programming languages [16–20], Brainstorm proposes four dynamic optimizations with statistical analysis of *Cell*-level dataflow: (1) by analyzing the number of *Cells* routed to branches, horizontally fuses multiple branches with GPU kernels optimized for frequent *Cell* loads; (2) with cross-layer *Cell*-level analysis, optimizes distributed placement of parallel branches to minimize inter-GPU communication; (3) with branch activation profiles, speculatively launches branch operators to hide routing overhead; and (4) speculatively preloads branch weights to save GPU memory.

We implement Brainstorm based on PyTorch by extending it with *Cell* and *Router*. We have implemented 6 state-of-the-art dynamic NNs using Brainstorm’s APIs, which are extensively evaluated on Nvidia GPUs. With the proposed dynamic optimizations, our evaluation shows Brainstorm achieves up to  $11.7\times$  speedup ( $3.29\times$  on average) or reduces memory consumption by 42%, compared with state-of-the-art solutions. We open-source Brainstorm to encourage more optimizations for dynamic NNs<sup>1</sup>. The key contributions are as follows.

- We identify a new space of optimization for dynamic NNs by leveraging the statistical profiles of dynamism to specialize model execution to runtime dynamism distribution.
- We identify the major challenge of optimizing dynamic NNs in existing DL frameworks is the misaligned granularity between the tensor-level programming and the fine-grained dataflow required to trace.

<sup>1</sup>Code available at <https://github.com/Raphael-Hao/brainstorm>

- We unify the programming of dynamic NNs with *Cell* and *Router* abstraction, making dynamism easy to trace.
- We propose multiple dynamic optimization strategies, leveraging the *Cell*-level dataflow analysis, which are shown effective for popular dynamic NNs.

We explain background and motivation in §2. We introduce Brainstorm’s key abstraction in §3. Four dynamic optimizations are proposed in §4. We present Brainstorm’s *Cell*-level dataflow analysis in §5. We explain the implementation in §6. We show the evaluation results in §7. We discuss handling distribution drift and other opportunities in §8. We compare with related works in §9. We conclude this paper in §10.

## 2 Background and Motivation

**Dynamic Neural Networks.** To mimic how the human brain works, the machine learning community actively works on how dynamic NNs should be designed. Various types of dynamism have been proposed to adapt the model structures and parameters to different inputs. Figure 1 illustrates representative patterns of dynamic NNs. The most common way of building a dynamic NN is to adaptively dispatch (parts of) inputs to different sub-networks with a routing mechanism. A common functionality, referred to as a *router* in this work, predicts which sub-network the input values should go through. Many routing policies have been proposed for different tasks, e.g., top-k router [3]. Sub-networks in different branches could have different weights, architectures, or the number of parameters to better fit the routed inputs. For example, MoE networks train parallel experts and dispatch input tokens into different expert sub-networks, each of which is expected to specialize in certain input categories [14, 15, 21, 22]. ClassSR [10] routes image patches to heterogeneous branches based on super-resolution difficulty. Skip-Conv [23] routes new pixels to computation and skips duplicated pixels of previous frames. Model developers often use a tensor to store multiple tokens/patches/pixels, and program sub-tensor dynamism using data movement operators like einsum [24].

**Dynamic optimization opportunities.** It has been widely studied in programming languages [25, 26] to leverage statistical profiles of program dynamism for just-in-time (JIT) optimization, e.g., HotSpot JVM speculatively trims paths never executed in collected runs [16]. However, optimizations in existing DL frameworks mainly focus on static NNs. They miss a lot of dynamic optimization opportunities brought by neural network dynamism.

Figure 2 illustrates routing distribution of four dynamic NNs. Figure 2a and Figure 2b are two dynamic NNs dispatching tokens and patches to different branches, respectively. We observe their distribution of tokens/patches is imbalanced: some branches receive non-negligibly more data than others. They have opportunities to tune efficient GPU kernels to

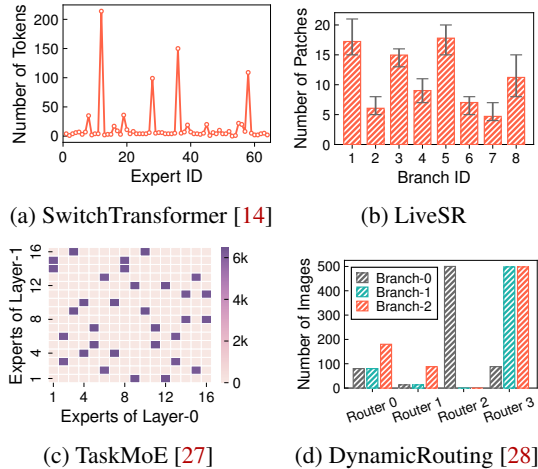


Figure 2: Distribution of routing in four dynamic NNs.

fit their shapes to load distribution, which could potentially bring over  $10\times$  speedup. Also, these parallel branches can be horizontally fused for concurrent execution (§4.1).

We also identify optimization opportunities by analyzing statistics of multi-layer correlation. Figure 2c illustrates the multi-layer correlation of TaskMoE [27], which is the portion of tokens from an expert at Layer-0 routed to another expert at Layer-1. We find the branch activation of two consecutive layers is correlated, e.g., it has a high probability that Expert-14/15 of Layer-1 will be activated after Expert-0 of Layer-0. Up to 87% of inter-GPU communication can be saved by co-locating correlated experts on the same GPU (§4.2).

Figure 2d shows branch activation of selected routers from DynamicRouting [28], which has 186 routers trained to forward images to one or two branches among three branches. Our measurement shows it spent over 44% time on routing. However, many routers have a biased distribution that tends to activate the same branch at different runs. E.g., Router-3 has a high probability of choosing Branch-1 and Branch-2. They create an opportunity for speculative execution, e.g., skipping routing computation to reduce routing overhead (§4.3), or opportunistically preload weight to GPU memory (§4.4).

Moreover, we find many dynamic NNs can be optimized by multiple dynamic optimizations simultaneously. The key requirement of these optimizations is the ability to collect statistical profiles at the granularity where dynamism happens, which is not explored by existing DL frameworks.

**Misaligned programming model.** The misaligned programming model is the major obstacle to tracing dynamism profiles in existing frameworks. As shown in Figure 1, language tasks typically route at the granularity of tokens from input sentences; vision tasks route patches from input images; video models partially reuse previous pixels depending on inter-frame similarity. All the dynamism happens inside the tensor of sentences, images, or frames. Existing frameworks

optimize models with a static dataflow graph, which expresses only the relation of tensors and operators. They have no ability to collect necessary profiles at runtime. Without explicit specification by model developers, they cannot understand what tokens are and how they are dynamically dispatched, let alone trace the complex token-level dataflow as Figure 2c requires. Moreover, tensor-level programming can only be applied with operator-level optimization (e.g., operator fusion) without the ability to optimize more fine-grained data movement or computation. These challenges motivate Brainstorm to propose a principled design to let model developers expose the information that needs to be traced and leverage the collected profiles for dynamic optimizations.

### 3 Cell and Router as the Core Abstraction

For model developers to express dynamic NNs in a traceable manner, Brainstorm unifies the model expression with *Cell* and *Router* to build dynamism at the correct granularity.

**Cell.** To let model developers define the data granularity where dynamism happens, Brainstorm augments a traditional tensor with a data abstraction called *Cell*. The *Cell* is the basic unit to be dynamically dispatched among multiple branches. Model developers can annotate any tensor using the `brt.annotate_cell` API to specify the granularity of *Cells* in a tensor (`brt` is the package name of Brainstorm).

```
brt.annotate_cell(tensor, dims, shape)
```

Model developers need to specify the values in which dimensions (`dims`) and which shape (granularity) to route. Figure 3 shows three examples that route values in *Cells* at the granularity of token, patch, and pixel, respectively. The first example routes a tensor with three tokens located at the 0-th dimension (`dims=(0)`), each represented by a vector of 768 float values (`shape=(1, 768)`). The second and third examples route  $32\times 32$  patches (`shape=(32, 32)`) and  $1\times 1$  pixels (`shape=(1, 1)`) in a 2D image tensor (`dims=(0, 1)`).

**Router.** To dynamically dispatch *Cells*, Brainstorm introduces a unified *Router* API that supports customized rules via `router_fn` to decide the dynamic placement of *Cells* among multiple branches. The API definition of *Router* and `router_fn` are elaborated as follows<sup>2</sup>.

```
class Router:
    def __init__(router_fn : Func)
    def forward(x : Tensor, kwargs) : Tuple[Tensor], Routes

def router_fn(x : Tensor, kwargs) : Routes
```

When initializing a *Router*, the `router_fn` should be specified to define the routing rule, i.e., how *Cells* should be routed among multiple branches. The `router_fn` takes the tensor

<sup>2</sup>We only show routing *Cells* of a single tensor. Multi-tensor routing has similar APIs, which are omitted due to the limited space.

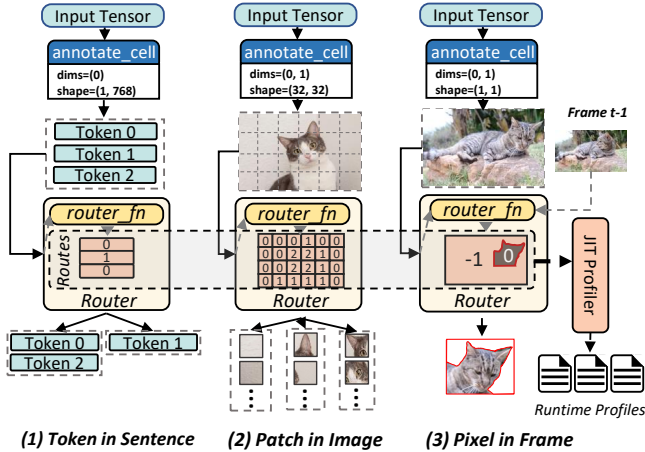


Figure 3: Examples of routing *Cells* at token-level, patch-level, and pixel-level. `router_fn` generates routing decisions indicating branch IDs should *Cells* be routed to (-1 for dropping), collected by the JIT profiler for dynamic optimization.

annotated with *Cells* as inputs and generates a special tensor *Routes*, whose value indicates which branch should *Cell* go. The shape of *Routes* has the same layout as *Cells* of the source tensor to route. E.g., the second example in Figure 3 has  $6 \times 4$  patches, thus `router_fn` should also generate  $6 \times 4$  *Routes*. Auxiliary inputs can be set in `kwargs` when making routing decisions. In the forward process of a model, *Router* feeds the input tensor to `router_fn` to get routing decisions for *Cells*, then dispatch *Cells* to corresponding branches. It is easy to port existing code of dynamic NNs to Brainstorm, e.g., we modify only 12 lines of code to port the official PyTorch implementation of SwitchTransformer [14] to Brainstorm.

Brainstorm’s *Router* abstraction decouples control-flow of deciding how *Cell* should be dynamically dispatched from its execution. Depending on runtime profiles, the optimal execution strategy varies greatly. Brainstorm eases model developers from challenging execution optimizations. They only need to focus on designing routing logic and leave execution optimizations to Brainstorm. The *Routes* given by `router_fn` are collected by JIT Profiler to get statistical profiles. Brainstorm’s dynamic optimizations analyze these statistics to find the most efficient execution strategy (§4).

Behind *Router* are a series of efficient GPU operations to realize the routing actions specified by `router_fn`. When branches receiving *Cells* are located on the same GPU, Brainstorm uses an efficient data rearrangement GPU kernel to generate multiple tensors containing *Cells* routed to each branch. Unlike existing solutions that heavily use computation operators (e.g., `einsum`) for fine-grained dynamic data rearrangement, Brainstorm uses a GPU kernel to directly move data to avoid unnecessary computation. When *Cells* are distributed to multiple GPUs, Brainstorm has a sparse communication primitive to efficiently scatter and gather *Cells*. Compared with the commonly used all-to-all primitive in existing DL

	Single layer Cell Loads	Multi-layer Cell Correlation	Branch Activation
Dynamic Horizontal Fusion	✓		
Profile-Guided Placement	✓	✓	
Speculative Routing			✓
Speculative Preloading			✓

Table 1: The statistical information used by different dynamic optimization strategies in Brainstorm.

frameworks [22, 29], Brainstorm’s sparse communication is more efficient when *Cells* are routed unevenly to multiple GPUs because it avoids unnecessary communication due to padding (refer to §6 for implementation details).

**Comparison to IR with control-flow.** Different from intermediate representations (IR) of existing DL frameworks that mix control-flow and dataflow together, Brainstorm chooses a decoupled design with *Router*. Brainstorm’s dataflow graph hides complex control-flow of *Router* behind `router_fn`. A *Router* can be regarded as a data distribution operator dynamically dispatching *Cells* of tensors to multiple branches. This greatly eases the tracing and analysis of *Cell*-level dataflow because compilers no longer need to separate dynamism-related operators from dataflow graphs, which is hard for DL frameworks [30–32]. Actually, instead of knowing how routing logic is constructed, it is more useful for compilers to know statistical information about routing decisions, which is sufficient to be captured by Brainstorm’s *Router*.

Moreover, Brainstorm further enhances control-flow operators in existing IR with *Cell*-level routing ability. Brainstorm’s *Router* itself can be regarded as a switch-case operator to route *Cells* to different branches for conditionally applying different functions. Together with a while-loop operator, a dynamic NN can route some *Cells* back to loop entry for the next iterations, and drop others to the output, which is commonly used by auto-regressive decoding of language tasks.

## 4 Dynamic Optimizations

Brainstorm analyzes the collected program execution profiles to improve runtime performance. Different from traditional dynamic optimization that analyzes the invocation of program functions or code blocks, the key for optimizing dynamic NNs is to profile and analyze *Cell*-level dataflow to specialize model execution to runtime dynamism distribution. In this section, we introduce four dynamic optimizations we identified for dynamic NNs. More optimizations are possible with Brainstorm’s *Cell* and *Router* abstraction. Table 1 lists the required information to conduct each dynamic optimization.

### 4.1 Dynamic Horizontal Fusion

Horizontal fusion is a compiler optimization to fuse concurrent branches of a model into a fused operator to improve



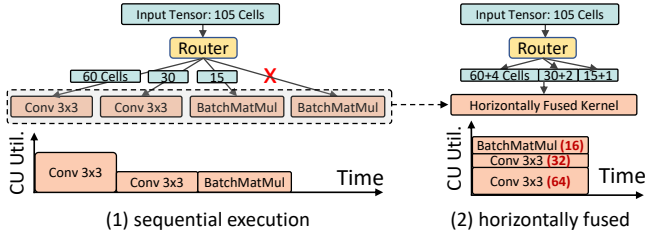


Figure 4: Operators of multiple branches are horizontally fused into one kernel. Only activated operators are executed. Each branch uses the nearest tuned kernel for least padding.

GPU Compute Unit (CU) utilization and reduce launching overhead. Existing approaches [33, 34] cannot be applied to dynamic NNs, because they assume a static dataflow graph whose branches are all activated with the same input. Brainstorm introduces a dynamic horizontal fusion optimization that supports dynamically and sparsely activated branches so that they can be executed on GPU simultaneously.

Especially, as we have shown in Figure 2, the *Cell* distribution can be very imbalanced for dynamic NNs. Even for large batch size, it can still accelerate the model execution by dynamical horizontal fusion of branches receiving a few numbers of *Cells*. Brainstorm leverages the profiles collected from *Router* to extract the statistical loads of each branch, i.e., how many *Cells* are routed to each branch. Brainstorm finds multiple percentiles (e.g., 50%, 90%, 100%) of the *Cell* load distribution, and tunes GPU kernels for these shapes. All tuned kernels are fused into one operator. At inference, Brainstorm pads the input of each branch to the nearest tuned kernel. This requires the traceability of the dynamic *Cell*-level dataflow at runtime that we explain how Brainstorm achieves it in §5.2. Note that the dynamically fused GPU kernel only uses the weights of activated branches without needing to load the weights of all branches into the GPU memory.

Figure 4 shows an example of routing 112 *Cells* among four parallel branches. Only three of the branches (only known at runtime) are activated. Before horizontal fusion, the three activated branches have to be executed sequentially, which may not saturate the GPU CU utilization. After fusing all branches into one GPU kernel, GPU can execute the activated branches simultaneously at a higher CU utilization. Each branch is executed with the tuned kernel of the least padding for the most efficient execution. For example, the fused kernel contains two tuned kernels of Conv 3x3 for 32 *Cells* and 64 *Cells*, which is used by the first two Conv 3x3 branches in the network by padding 4 and 2 *Cells*, respectively.

## 4.2 Profile-Guided Model Placement

The cerebral cortex of human brain is organized into distinct areas, whose neurons of a function are located closely [35]. By analyzing statistical routing decisions, we observe similar effects in artificially designed dynamic NNs. As shown in Fig-

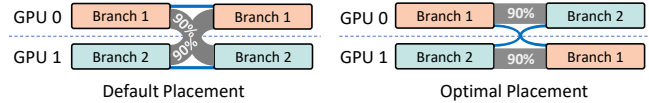


Figure 5: Profile-guided model placement. The example shows the default placement has 90% inter-GPU traffic, which is reduced to 10% by the optimized placement.

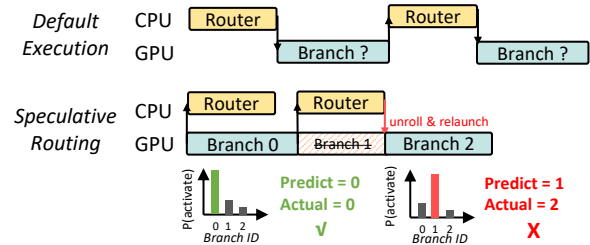


Figure 6: Speculative routing: skip routing computation and speculatively launch the branch w/ highest probability; unroll and relaunch to correct branches when mispredicted.

Figure 2c, experts from two layers are activated together with a high probability. The *Cell*-level communication between these highly-correlated experts is higher than the others. Figure 5 illustrates an example that, by analyzing the multi-layer correlation, Brainstorm can co-locate correlated sub-networks on the same GPU to reduce inter-GPU communication. Note that, in addition to dynamic *Cell*-level dataflow collected at runtime, the multi-layer correlation also needs to analyze static *Cell*-level dataflow to infer correct placement constraints. Our analysis in §5.1 shows each *Cell* of a sentence tensor depends on all *Cells* from the previous MoE layer. This implies a placement constraint that all *Cells* of a sentence should be gathered at the same GPU so that its self-attention operator can generate correct outputs. This presents a challenge requiring both dynamic and static *Cell*-level dataflow analysis to understand the inter-layer correlation of *Cells*. We explain Brainstorm’s static *Cell*-level dataflow analysis in §5.1.

In addition to cross-layer analysis, we find single-layer *Cell* distribution like Figure 2a can also help model placement. Some branches could take more *Cells* than others. Heavy branches can be co-located with light branches to balance the overall communication to avoid stalling on some GPUs.

## 4.3 Speculative Routing

Model developers often build routing logic involving control flows, which may require CPU processing and incur CPU-GPU synchronization overhead. Compared to their theoretical performance (based on FLOPs), routing overhead may dominate the inference latency. Our measurement shows MS-DNet [1] and DynamicRouting [28] spend 65% and 44% time in routing. We find these model often has a biased probability when selecting branches at inference. Our analy-



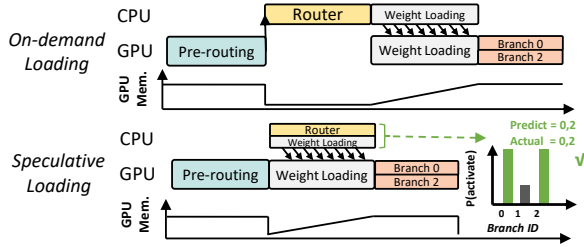


Figure 7: Speculatively preload weights with the highest probability; fallback to on-demand loading when mispredicted.

sis of Brainstorm’s *Router* profiles shows many *Routers* are highly predictable. Brainstorm can predict the decisions of DynamicRouting [28] with an accuracy over 90% by just choosing the most frequently appeared branches (§7.4.6). As Figure 6 shows, Brainstorm can predict the routing decisions of *Routers* in advance (based on statistical profiles) and skip `router_fn` to hide the routing overhead. To guarantee the correctness, Brainstorm uses a parallel thread to check the result of `router_fn`. When misprediction happens, the model execution will be unrolled to re-execute the correct branch with negligible misprediction overhead (§7.3).

#### 4.4 Speculative Weight Preloading

To run inference of a large model on a limited size of GPU memory, it often requires swapping weights of layers between GPU memory and host memory to reduce the GPU memory requirement [36]. To hide the memory migration latency, existing solutions need to know the execution order of layers to preload necessary weights while executing previous layers in a pipelined manner [37, 38]. However, dynamic NNs do not have a static order of layer execution. The execution of dynamically activated branches is only known when the routing decisions are made. This makes it hard for existing solutions to preload weights of dynamic layers. As shown in Figure 7, similar to speculative routing, Brainstorm leverages the statistical profiles of branch activation distribution to speculatively preload weights of branches that can be activated with a high probability. It falls back to on-demand loading with negligible overhead (§7.3) when the predictive preloading misses.

### 5 Tracing Cell-level Dataflow

To realize optimizations in §4, it is important to understand how *Cells* are transmitted along a network so that the compiler can leverage the *Cell*-level dataflow to optimize model execution. In dynamic NNs, there are two types of *Cell*-level dataflow: (1) static dataflow existing in most static operators (e.g., Conv2D), which is fixed for all inputs; and (2) dynamic dataflow, which is determined by *Routers* at runtime. The former is to understand *Cell*’s relationship across static layers; the latter is to identify the *Cell* routing among branches.

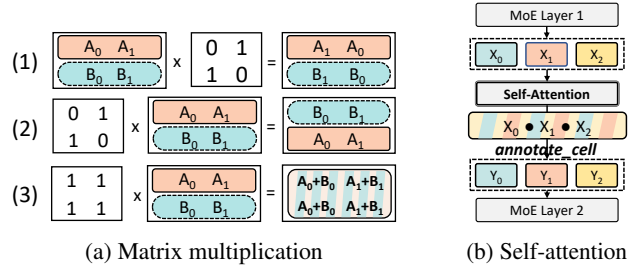


Figure 8: Different types of static dataflow at *Cell*-level.

#### 5.1 Static Cell-level Dataflow

Tensor-centric dataflow graphs only preserve relations between tensors without the information of *Cells*. To trace all possible *Cell*-level dataflow of static operators, Brainstorm uses symbolic execution at *Cell*-level to extract finer-grained relations in ahead-of-time compiling. With the annotated *Cells* of a tensor, Brainstorm initializes a symbolic version of the tensor, whose *Cells* are symbols. Tensor values belonging to one *Cell* share the same symbol. Brainstorm leverages the *tensor expression* of operators (widely used in DL compilers [39]) to build computation logic of operators. By checking the results of symbolic computation, Brainstorm understands how *Cells* are transmitted in static operators.

Figure 8a illustrates three examples of matrix multiplication between a tensor of multiple *Cells* and a constant matrix. The tensor has two *Cells* annotated as *A* and *B*. The first preserves *Cell* positions; the second reorders *Cells*; the third mixes all *Cells* in the output. This example shows the static *Cell*-level dataflow could vary when the tensor values are different. It is hard for tensor-level dataflow analysis to obtain this finer-grained relation. Figure 8b demonstrates the static *Cell*-level dataflow of the self-attention operator between two MoE layers. Because there is a matrix multiplication between two tensors in the self-attention operator and both tensors contain *Cells* of  $X_i$ , this self-attention operator mixes all *Cells* from input  $X$  to generate the output  $Y$ . With symbolic execution of *Cells*, we can derive the relations between the *Cells* in  $X$  and  $Y$ , i.e., every *Cell* in  $Y$  is derived from all *Cells* in  $X$ .

The static *Cell*-level dataflow analysis is necessary to derive cross-layer relations of *Cells*, which is important in data movement-related optimization. It allows Brainstorm to explore data movement at the *Cell*-level, breaking the limitation of tensor-level data movement when optimizing multi-GPU execution. For example, if *Cells* are only reordered without mixing (e.g., the first two types in Figure 8a), the framework has more freedom to dispatch *Cells* among multiple GPUs based on their data locality for better performance. For MoE-based models, because the tokens are mixed up in the self-attention layer, it introduces a constraint that requires aggregating all tokens of a sentence to the same GPU before self-attention to derive the output. As we have shown in §4.2, this requirement creates constraints of how *Cells* should be

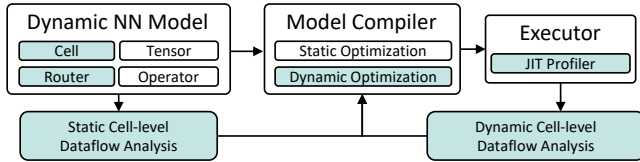


Figure 9: The system architecture of Brainstorm. Shaded components are introduced by Brainstorm for dynamic NNs.

dynamically placed in optimization, which is only known after the static *Cell*-level dataflow is analyzed.

## 5.2 Dynamic *Cell*-level Dataflow

In Brainstorm, model developers express dynamism using *Router*. The routing logic is defined in `router_fn`, which generates routing decisions of *Cells* at runtime. Brainstorm’s *Router* abstraction makes it easy to trace the necessary information. Similar to dynamic optimization of traditional programming languages, Brainstorm focuses on collecting statistical profiles of routing decisions without caring about how they are generated.

If *Cell*-level profiling is enabled, when each time a *Router* is called, Brainstorm records its routing decision into a buffer. Brainstorm has a separate thread to stream the buffer to a profile file. Brainstorm supports multi-level profiling. Some optimizations only require local statistical profile of *Router* (e.g., branch load of *Cells*). Some optimizations require *Cell*-level dataflow across multiple layers, thus needing to dump raw decisions directly. As control signals, routing decisions are much smaller than other data tensors in dynamic NNs. Our evaluation in §7.3 shows the profiling overhead is negligible.

## 6 Implementation

We implement Brainstorm on Pytorch with 13,000 LOC: 3,000 lines for Brainstorm core abstraction, 3,000 lines for dynamic optimizations, 3,000 lines of C++ code for kernel scheduling and sparse *Cell* communication, and 1,500 lines for auto-transformation to support dynamic optimizations.

Figure 9 summarizes Brainstorm’s architecture. In addition to widely-used Tensor and Operator in existing frameworks, Brainstorm introduces *Cell* and *Router* to express dynamic NNs in a unified abstraction (§3). The programmed dynamic NNs will be optimized by the compiler with both static and dynamic optimizations (§4). Brainstorm’s dynamic optimization needs both static and dynamic *Cell*-level dataflow analysis (§5). Brainstorm first infers the static *Cell*-level dataflow in static operators (§5.1) in an ahead-of-time manner. When executing the compiled model, a JIT profiler collects *Router* profiles for further dynamic dataflow analysis (§5.2).

**Efficient *Cell* routing.** Brainstorm is responsible for dy-

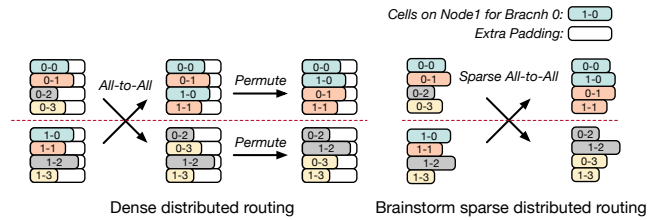


Figure 10: Sparse All-to-All for distributed *Cell* routing. It saves redundant communication of extra padding.

amic *Cell* dispatching that is aware of dynamic optimization applied, leaving model developers to focus on designing the routing algorithm. For *Cell* routing on a single GPU, we use a custom GPU kernel to rearrange *Cells* inside a tensor according to the routing decisions. We borrow the idea from Tutel [22] for MoE models by rearranging *Cells* for all branches in parallel with a custom GPU kernel. But our implementation is general to all dynamic NNs in addition to MoE models. Moreover, our implementation is aware of the dynamic optimization applied. For instance, a dynamic horizontal fused operator may contain GPU kernels of varied sizes, thus requiring variable padding. For *Cell* routing across multiple GPUs, we provide a more flexible sparse communication primitive. As shown in the left of Figure 10, model developers often combine dense all-to-all primitive and permutation operations for distributed *Cell* routing. Its efficiency is restricted to balanced routing. With Brainstorm’s sparse communication, it only transmits *Cells* without extra padding. The underlying implementation of sparse communication is a collection of point-to-point communication. However, it can adapt to the dynamic optimization’s requirements and provide the most efficient communication mechanism.

**Excessive Candidates for Kernel Fusion.** Brainstorm fuses multiple branches into one kernel function, each comprising several potential candidates. At runtime, Brainstorm triggers suitable candidates based on the dispatched *Cells*. However, excessive kernel candidates derived from profiling analysis can lead to considerable time overhead when searching for them using auto-tuning tools [39]. To avoid issues in this case, Brainstorm only fuses a limited set of candidates of each branch. Meanwhile, kernel candidates are shared between branches if the fused branches are homogeneous (the same operator only with different weights). For instance, since SwitchTransformer uses the same feed-forward layer for its experts, Brainstorm only needs six candidate kernels to optimize the execution of 256 experts per layer (§7.4.1).

**Optimization Passes.** Most automatic transformations in Brainstorm are implemented with `torch.fx`. With the dataflow graph traced by `torch.fx`, Brainstorm uses the statistical profiles collected from *Routers* to manipulate the dataflow graph for optimization. E.g., in dynamic horizon-

Model	Dataset	Fusion	Place	Route	Load
Switch [14]	MNLI [40]	✓			
TaskMoE [27]	Synthetic		✓		
SwinV2-MoE [41]	ImageNet22k [42]		✓		
LiveSR	Iowa-DOT [43]	✓			
DRouting [28]	Cityscapes [44]			✓	✓
MSDNet [1]	Imagenet [42]	✓		✓	

Table 2: Benchmark specifications. (Fusion: Dynamic Horizontal Fusion; Place: Profile-Guided Placement; Route: Speculative Routing; Load: Speculative Weight Preloading.)

tal fusion, we replace operators of multiple branches in the dataflow graph with the generated fused kernel of multiple shapes and change *Router* to pad tensors to supported shapes while routing *Cells*. For speculative routing, we reorder operators in the dataflow graph to skip and unroll routing logic. For speculative weight loading, we collect parameters of branches, and insert extra operators for loading and unloading them at runtime. The profile-guided model placement is an exceptional case, as the loading of model weights falls outside the scope of `torch.fx`. Before inference, Brainstorm loads corresponding weights given by the placement plan derived from the statistical profiles. At runtime, Brainstorm’s *Router* will translate routing decisions given by `router_fn` according to the placement to route *Cells* to the appropriate devices.

**Selecting Dynamic Optimizations.** Given a dynamic model, we use a rule-based policy to select dynamic optimizations. Dynamic horizontal fusion is used for models with parallel branches when a single branch cannot saturate GPU cores. Profile-Guided model placement is used for multi-GPU inference. Speculative routing and weight preloading are enabled when routers block GPU kernel submission. Speculative weight preloading is used when GPU memory is limited, and paging is used. Table 2 has listed the dynamic optimizations applied to each model. For example, LiveSR is a lightweight super-resolution model, and a single branch may not saturate a GPU. Thus we apply dynamic horizontal fusion to it. Also, MoE-based models are usually large language models requiring multi-GPU deployment. Therefore we apply the placement optimization. The input for TaskMoE is sufficiently large for high GPU utilization, thus no need for horizontal fusion.

## 7 Evaluation

We evaluate the performance of Brainstorm (BRT) on six representative dynamic NNs. We compare Brainstorm with various approaches to execute and optimize dynamic NNs, including PyTorch-native static optimizations and model-specific optimizations (e.g., Tutel for MoE). Overall, Brainstorm achieves up to  $11.7\times$  speedup ( $3.29\times$  on average) or reduces GPU memory usage by 42%.

Model	Switch	TaskMoE	SwinV2-MoE
LOC	12	24	14
Model	LiveSR	DRouting	MSDNet
LOC	6	18	14

Table 3: Lines of code for porting the model to Brainstorm.

## 7.1 Experimental Setup

**Testbed.** We evaluate Brainstorm with two separate setups for single-GPU and multi-GPU experiments. The single-GPU evaluations use a server with AMD-EPYC-7V13 CPUs and one NVIDIA A100 (80GB) GPU running CUDA 11.3 and cuDNN 8.6. The multi-GPU evaluations use a server with Intel Xeon CPU E5-2690 v4 CPU and *eight* NVIDIA V100 (32GB) GPUs running CUDA 11.3 and cuDNN 8.2.

**Benchmarks and datasets.** Our evaluations are performed to run inference of six representative dynamic NNs, covering vision and natural language processing (NLP) tasks. Table 2 lists evaluated models, datasets, and dynamic optimizations we apply in Brainstorm. SwitchTransformer (Switch) [14] and TaskMoE [27] are two MoE models for NLP, whose *Cells* are defined at token level and sentence level, respectively; SwinV2-MoE [22, 41], LiveSR, DynamicRouting (DRouting) [28], and MSDNet [1] are four models for vision tasks. SwinV2-MoE and LiveSR define a *Cell* at image patch level. DynamicRouting and MSDnet use an image as a *Cell*. Statistical profiles used for Brainstorm’s dynamic optimizations are collected from training datasets and evaluated in test datasets.

**Baselines.** We mainly compare Brainstorm with PyTorch and Tutel in all experiments. As far as we know, PyTorch is a state-of-the-art framework that can flexibly support dynamic neural networks (thanks to the expressiveness of Python). The official implementation of all models we evaluated are based on PyTorch and thus are compared in all evaluations in this paper. Tutel is designed specifically for MoE. Thus we only compared Brainstorm with Tutel on MoE-based models. To evaluate the benefit of the new proposed dynamic optimizations, Brainstorm and all baselines use the same static optimizations (e.g., vertical kernel fusion) in all experiments.

## 7.2 Effectiveness of Brainstorm Abstraction

**Expressiveness of Brainstorm.** Brainstorm’s abstraction can express various dynamic neural networks in a simple and concise manner. Table 3 shows the lines of code for porting the six dynamic neural network models to Brainstorm. Brainstorm unifies the API of expressing routing logic through *Router* and *Cell*. This only adds a marginal extra coding effort to porting existing models and building new dynamic models. Brainstorm eases the programming by providing common

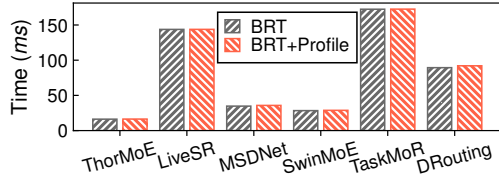
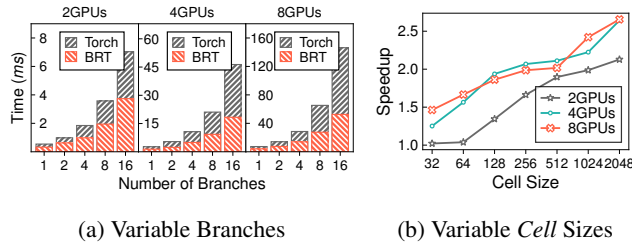


Figure 11: Latency of Brainstorm with or without profiling.



(a) Variable Branches (b) Variable Cell Sizes

Figure 12: Performance comparison of sparse all-to-all between PyTorch and Brainstorm.

router\_fns (e.g., Top-K) and allows model developers to construct more complex ones atop them.

**Overhead of Tracing Dynamic Cell-level Dataflow.** This micro-benchmark presents the overhead of tracing dynamic Cell-level dataflow. Figure 11 shows the latency variation when tracing is on and off. The latencies of all models are almost equal before and after tracing is enabled. The average overhead is less than 1.0% for all models.

When routing actions are calculated at GPU, major overhead comes from GPU kernels for statistics. The synchronization overhead is negligible because Brainstorm dumps profiles to the CPU periodically and asynchronously.

**Effectiveness of Cell Routing.** Brainstorm’s Router decouples routing logic from execution. Brainstorm has efficient implementations to conduct dynamic data movement for sparse communication. Figure 12 demonstrates two micro-benchmarks for sparse communication, which is a multi-gpu experiment. We randomly generate 1024 Cells routed from one GPU to multiple GPUs. Figure 12a measures the latency of PyTorch’s all-to-all collective (nccl [45] as backend) and Brainstorm’s sparse communication with varied numbers of branches and GPUs. Each Cell has 512 Float32 values (same as TransformerBase [46]). Brainstorm achieves 1.88× to 2.78× speedup from 2 to 8 GPUs. Figure 12b shows Brainstorm’s speedup with a varied Cell size from 32 to 2048 Float32 values, with 4 branches on each GPU. Brainstorm achieves 2.13× to 2.66× speedup on 2 to 8 GPUs. Overall, Brainstorm performs better than PyTorch in all experiments. The root cause is the extra communication for padding using PyTorch’s all-to-all communication, which is avoided by Brainstorm’s sparse communication.

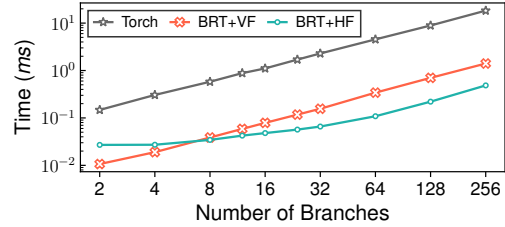


Figure 13: Latencies of serial execution, vertical fusion, and dynamic horizontal fusion with variable branches

### 7.3 Micro Benchmarks

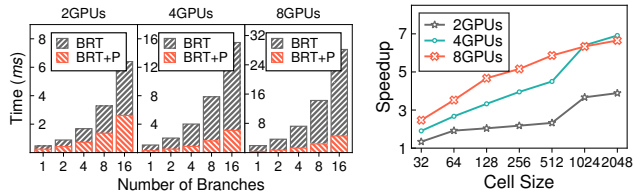
**Dynamic Horizontal Fusion.** In the micro-benchmark of Brainstorm’s dynamic horizontal fusion, we build a simple multi-branch network, each of which contains a Conv2D operator. A Router dispatches 32x32 image patches to different branches based on image content. Brainstorm tunes kernels from 4 patches to 9 patches based on the collected Router profiles. It is conducted on the single-GPU server.

Figure 13 presents the latencies of PyTorch’s serial execution (Torch), Brainstorm’s serial execution but with tuned kernels (BRT+VF), and Brainstorm’s dynamic horizontal fusion (BRT+HF).

Vertical fusion (VF) is the commonly used fusion of consecutive operators to reduce kernel launching overhead [39, 47]. Compared to Torch, BRT+HF achieves up to 41.8× speedup. The improvement comes from two sources: the improved CU utilization with concurrent execution of multiple branches, and efficient kernels tuned for frequently appeared Cell loads. By comparing BRT+VF and Torch, we identify the statistically tuned GPU kernels that bring 13.1× speedup. The concurrent execution of multiple branches further brings 3.18× speedup (BRT+HF/BRT+VF). Since dynamic horizontal fusion has an overhead of extra GPU kernels to calculate input pointer addresses, we find BRT+HF performs slightly worse than BRT+VF (12.3μs on average) when the number of branches is small.

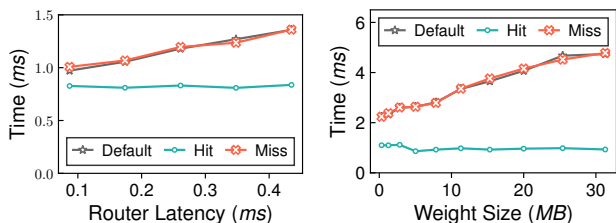
**Profile-Guided Placement.** In §4.2, we show that profile-guided model placement can save inter-GPU communication for dynamic NNs. In this micro-benchmark, we compare the communication latency of default placement in PyTorch with Brainstorm’s optimized placement. We conduct this experiment on the multi-GPU server. We replace PyTorch’s communication with Brainstorm’s sparse communication to isolate the improvement from efficient sparse communication. In the default placement, each GPU-*i* routes 1024 tokens to each branch on GPU-*(i + 1)* and 10 tokens per each other branch. In the optimized placement, Brainstorm can route 1024 tokens to the same GPU without inter-GPU communication. In Figure 14a, the Cell size is fixed to 512 Float32 values for evaluation with variable branches. Brainstorm achieves 2.45× to 6.23× speedup on 2 to 8 GPUs. In Figure 14b, the number





(a) Variable Branches (b) Variable *Cell* Sizes

Figure 14: Performance comparison of sparse communication of Brainstorm with (BRT+T) and without (BRT) the placement optimization.



(a) Variable *Router* latencies (b) Variable weight sizes

Figure 15: Performance comparison of default execution and hit/miss cases in speculative optimizations.

of branches on a single GPU is fixed to 4 with variable *Cell* sizes. Brainstorm achieves  $3.89\times$  to  $6.65\times$  speedup on 2 to 8 GPUs. Brainstorm achieves the improvement due to reduced communication in the optimized placement. More branches and larger *Cell* further increase inter-GPU communication volume amplifying the gap between the default placement and Brainstorm’s optimized placement.

**Hit or Miss of Speculative Optimization.** §4.3 and §4.4 introduce two speculative optimizations for dynamic NNs, i.e., speculative routing and weight preloading. The following two micro-benchmarks demonstrate a comparison between default execution and Brainstorm’s speculative optimization, conducted on the single-GPU server. We build a simple network routing an input tensor to 8 branches. Each branch has 20 `gemm` operators. Brainstorm speculatively executes Branch-0 or loads Branch-0’s weights in the speculative routing and weight preloading, respectively. Figure 15a shows inference time with varied *Router* latency. When prediction hits, *Router* latency can be hidden by `gemm` operators on the correct branch. When prediction misses, these `gemm` operators will be unrolled. Brainstorm achieves a constant inference time when prediction hits, and a similar inference time with the default execution when prediction misses.

Figure 15b shows weight preloading overhead of the same model but with varied weight sizes. Since only weights of the activated branch are loaded, the GPU memory requirement is reduced by  $8\times$ . When Brainstorm’s prediction hits, the

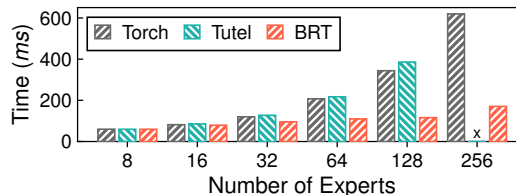


Figure 16: Latencies of SwitchTransformer.

weights are speculatively preloaded before *Router*, whose latency is hidden by previous computation and *Router* latency. When prediction misses, Brainstorm falls back to the default execution that loads weights of the correct branch. Brainstorm achieves a consistent latency when prediction hits, and similar latency with the default execution when prediction misses.

## 7.4 End-to-end Model Execution

### 7.4.1 SwitchTransformer

In SwitchTransformer, each expert has a capacity of 64 tokens for each sentence. By analyzing *Router* profiles, we find an imbalanced distribution of the number of tokens routed to each expert (shown in Figure 2a). This motivates us to apply dynamic horizontal fusion to execute experts in parallel with GPU kernels tuned for different loads. We use the official weights trained by Google with 8 to 256 experts per MoE layer. The batch size is 8, and each sentence has 128 tokens. The experiment is conducted on the single-GPU server.

Figure 16 shows latencies of SwitchTransformer with official implementation in PyTorch (*Torch*), replacing MoE layers with an optimized implementation from Tutel (*Tutel*), and Brainstorm with dynamic horizontal fusion. The official implementation executes experts in serial. Tutel runs experts concurrently with `BatchMatmul`, which requires padding to the same number of tokens for all experts. Brainstorm outperforms by  $3.63\times$ , and  $3.33\times$  compared to *Torch*, and *Tutel*, respectively. The speedup increases with more experts in each MoE layer. In addition to improved utilization of concurrently executed experts, Brainstorm also benefits from imbalanced token distribution. Because many experts only receive a few tokens, Tutel pads many dummy tokens in all paths, leading to vast wasted computation on padding. The excessive padding also uses more GPU memory leading to out-of-memory in Tutel when there are 256 experts. By analyzing loads of different branches, Brainstorm compiles multiple GPU kernels to minimize the padding.

### 7.4.2 LiveSR

LiveSR is our internal model for super-resolution, which slices a single image into  $32\times 32$  patches and routes them to different branches. It uses a ResNet-18 model to extract patterns, which are then routed by K-nearest neighbor (kNN) to multiple branches. By collecting the routing distribution of patches,

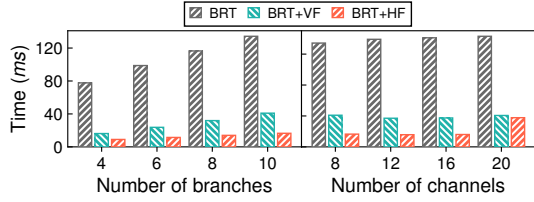


Figure 17: Latencies of LiveSR in Brainstorm with vertical fusion and dynamic horizontal fusion.

we find a distribution in the number of patches routed to each branch (as shown in Figure 2b). This creates the opportunity for Brainstorm to tune GPU kernels for frequently *Cell* loads. Also, since different patches of an image are routed to different branches, Brainstorm can horizontally fuse these branches to concurrently execute them to improve CU utilization.

Figure 17 shows latencies of LiveSR with different optimizations, while being executed on the single-GPU server. BRT+HF applies dynamic horizontal fusion of both multiple branches and multiple tuned kernels of different loads. To dissect the improvement of both types of fusion, we evaluate BRT+VF that only fuses Conv2D, BatchNorm, and ReLU operators in each branch but with statistically tuned GPU kernels. In Figure 17, we vary both the number of branches with a fixed number (8) of channels and the number of convolution channels with a fixed number (10) of branches.

Overall, BRT+HF achieves up to  $8.62\times$  speedup compared to BRT. BRT+VF brings a speedup up to  $3.5\times$  compared with BRT. BRT+HF further brings  $1.79\times$  to  $2.48\times$  gains over BRT+VF with an increasing number of branches because of the improved CU utilization with more branches. When increasing the number of channels, we find the latency of BrainstormBRT+HF remains the same until it reaches 20 channels as it goes beyond the upper bound of GPU CUs.

### 7.4.3 TaskMoE

TaskMoE routes input tensors at the granularity of the sentence. Each MoE layer has 16 experts. Each sentence is routed to 2 experts. The key difference of TaskMoE is its routing algorithm: it decides expert of a sentence based on *task type*. Sentences of the same task will be routed to the same expert branches. Therefore, as we have shown in Figure 2c, TaskMoE has a strong inter-layer expert correlation that experts of the same task are activated together with a high probability, which brings the opportunity for profile-guided placement.

Brainstorm optimizes placement by reordering experts of MoE layers for the most efficient communication. Brainstorm’s *Routers* are aware of reordering and dispatch sentences to correct GPUs in the optimized placement. We conduct this experiment with three input settings: 256 sentences on each GPU with 32/64 tokens in each sequence; 512 sequences on each GPU with 32 tokens in each sequence. The

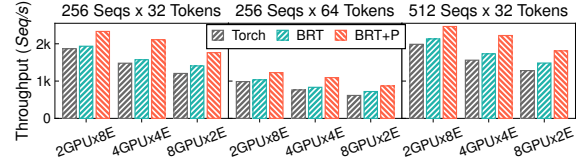


Figure 18: Throughput of TaskMoE. Torch: routing with PyTorch’s native communication primitive. BRT: routing with Brainstorm’s sparse communication primitive. BRT+P: placement optimized routing over BRT.

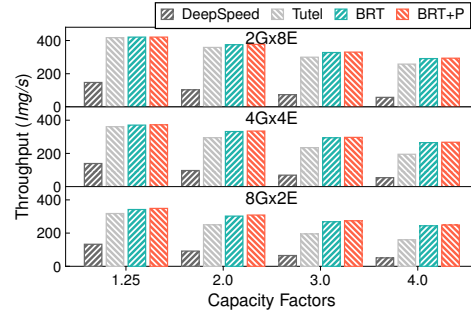


Figure 19: Throughput of SwinV2-MoE. G: the number of GPUs. E: the number of experts per GPU.

task ID of each sequence is randomly generated. Since routing of TaskMoE only works on task ID, the synthetic dataset does not affect the optimal placement and evaluation conclusion.

Figure 18 shows the per-GPU throughput on 2-8 GPUs. The experiment is conducted on the multi-GPU server. Compared with Torch, BRT first brings up to  $1.17\times$  speedup with efficient sparse communication. The speedup of BRT grows with more GPUs because of the increased data volume for inter-GPU transmission. Brainstorm’s sparse communication saves unnecessary communication due to padding. On top of this, BRT+P further achieves up to  $1.34\times$  speedup with the optimized placement. The optimized placement derived from runtime profiles helps BRT+P to reduce  $42\sim 87\%$  inter-GPU communication, speeding up routing of MoE layers.

### 7.4.4 SwinV2-MoE

SwinV2-MoE is the MoE-version of SwinTransformer [41] for image tasks, introduced in Tutel [22]. It defines tokens as *Cells*, each of which contains  $384 \text{ float}_{32}$  values tokenized from a  $48\times 48$  image patch. SwinV2-MoE uses a capacity factor to control the number of patches each expert receives. When the capacity is exceeded, extra patches are dropped during routing. The capacity factor varies in  $[1.25, 2.0, 3.0, 4.0]$  in the experiments. We evaluate SwinV2-MoE with 16 experts on the multi-GPU server by evenly placing the experts on each GPU with 128 images for each inference.

Figure 19 shows throughput of four approaches: a PyTorch

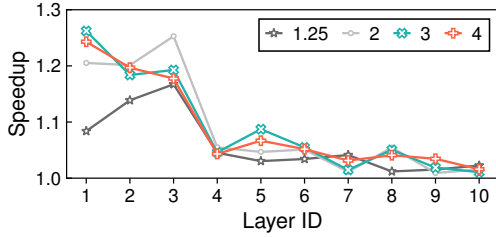


Figure 20: Gaps between the *best* and the *worst* placement for MoE Layers in SwinV2-MoE with varied capacity factors.

implementation using DeepSpeed-MoE [48] (DeepSpeed), optimized version with Tutel’s MoE kernels [22] (Tutel), optimized version with Brainstorm’s Router (BRT), and Brainstorm’s profile-guided placement optimization (BRT+P). Brainstorm’s efficient Router first brings up to  $5.04\times$  and  $1.52\times$  speedup over DeepSpeed and Tutel, respectively. Both BRT and Tutel use custom GPU kernels for efficient routing inside a GPU, thus greatly outperforming DeepSpeed, which uses `einsum`. With an increased capacity factor, BRT brings higher speedup over Tutel because of saved inter-GPU communication due to increased padding.

By optimizing expert placement via runtime profiles, we find BRT+P only brings marginal improvement. After using Brainstorm’s efficient Router, SwinV2-MoE model only spends up to 35% of time on inter-GPU communication, which reduces the potential by further reducing communication overhead. Similar to TaskMoE, we do observe different expert placements have greatly varied efficiency. Figure 20 shows our evaluation of a single SwinV2-MoE layer to compare the performance of the best placement and the worst placement with 8 GPUs and 2 experts per GPU. The gap is up to  $1.26\times$  speedup for ten SwinV2-MoE layers. The smaller the layer id is, the more imbalance appears in token distribution, creating more space for improvement by placement. It shows great potential for larger MoE models with more experts, whose communication latency dominates [22].

#### 7.4.5 MSDNet

MSDNet [1] is a dynamic network that can adapt this execution path to the computational resource limits at test time. The network contains 5 exits that allow the inference of an image to end in the middle, if the output quality is higher than the predefined thresholds. Users can configure the threshold of each exit to control the inference cost. For instance,  $[0, 0, 0, 0.4, 0.6]$  represents that 40% of the inferences in the dataset end at the 4th exit and 60% end at the last exit. There are no inferences ending at the other exits.

Figure 21 shows the experiment results with 6 kinds of exit configurations applying different optimizations, running on the single-GPU server. We set the batch size to a single image at inference. We first tune the GPU kernels with vertical fusion (BRT+VF) as the baseline. On top of that, we first

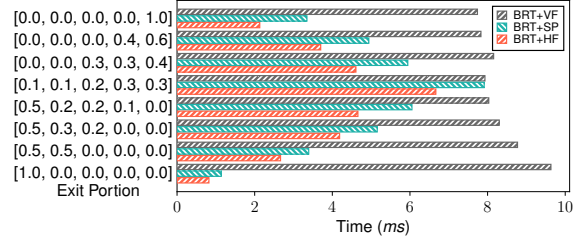


Figure 21: Latencies of MSDNet with vertical fusion, speculative routing, and dynamic horizontal fusion.

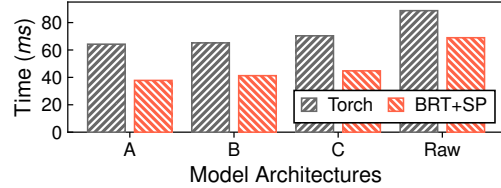


Figure 22: Latencies of DynamicRouting.

apply speculative routing (BRT+SP) and then dynamic horizontal fusion (BRT+HF) to evaluate the benefits of dynamic optimizations. Compared with BRT+VF, Brainstorm achieves up to  $8.44\times$ ,  $11.7\times$  speedup by BRT+SP and BRT+HF, respectively. We observe BRT+SP reduces higher latency when the inferences end at either very early exits or very last exits, due to the speculative routing making more correct predictions. If the inference has a similar opportunity to end at each exit, BRT+SP has a similar performance with BRT+VF (e.g., for  $[0.1, 0.1, 0.2, 0.3, 0.3]$ ). For dynamic horizontal fusion (BRT+HF), Brainstorm performs better when the inferences prefer ending at the last exits, further bringing up to  $1.57\times$  gain over BRT+SP. The root cause is the uncertain routers break many horizontal fusion opportunities. MSDNet has some operators that can be executed in parallel if the inference does not end at an exit. If a Router may terminate in the middle, Brainstorm cannot determine whether it is safe to horizontally fuse them, thus falling back to BRT+VF.

#### 7.4.6 DynamicRouting

DynamicRouting [28] is a semantic segmentation model for images that introduces a lot of Routers. It contains 186 Routers and 186 computation operators, leading to a very high routing overhead. At each Router, input images are routed to 1 or 2 branches among 3 designed branches with convolution operators for down-sampling, up-sampling, or keeping-resolution, respectively. DynamicRouting proposes four architecture configurations (A, B, C, and Raw for short, in order of growing computation). By analyzing Routers’ runtime profiles collected by Brainstorm, we find many Routers exhibit a high probability of making consistent routing decisions, which brings opportunities for speculative optimizations. The following experiments are conducted on the single-GPU server.

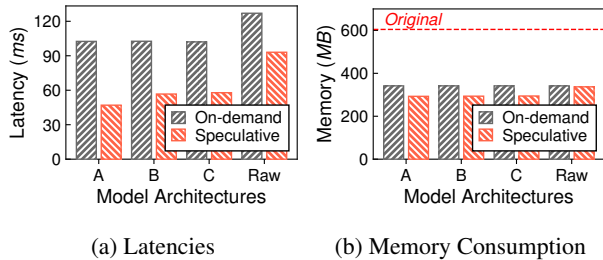


Figure 23: Speculative weight preloading of DynamicRouting with variable model architectures.

Figure 22 presents the latency of four configurations optimized by Brainstorm’s speculative routing (BRT+SP), where batch size is set to a single image. Brainstorm achieves up to  $1.7\times$  speedup compared to the official implementation in PyTorch (Torch). BRT+SP achieves  $1.7\times$ ,  $1.58\times$ ,  $1.57\times$ , and  $1.29\times$  speedup compared with Torch in the four architecture, respectively. With statistical distribution derived from the runtime profiles, BRT+SP can predict the routing decisions of the 186 routers with an accuracy of 90% ~ 95%. This greatly reduces the routing overhead in the four model architectures. As we have shown in the micro-benchmark of Figure 15a, the overhead of speculative routing is negligible even when the prediction is wrong.

Figure 23 shows the inference latency and the GPU memory usage of DynamicRouting optimized by Brainstorm’s speculative weight preloading. In the baseline (on-demand loading), Brainstorm only loads the weight of a branch after the routing decision is made. Brainstorm will preload the weights of the branch to be activated with the highest probability, and falls back to on-demand loading if the prediction is wrong. Because the weight loading latency is hidden, Brainstorm’s speculative optimization can accelerate the model inference by up to  $1.97\times$  than on-demand loading. Moreover, the official implementation needs to load all model weights to the GPU memory for single-image inference (i.e., 604.5MB of Original in Figure 23). With on-demand loading and speculative preloading, memory usage is greatly reduced by 50.7% and 43.5%, respectively. This creates the opportunity to infer large models on GPUs with limited GPU memory. Brainstorm’s speculative weight preloading requires slightly lower GPU memory than on-demand loading. This is because speculative weight preloading also releases some GPU memory in advance speculatively.

## 8 Discussion

**Handling distribution drift.** The profiling data is analyzed offline by dynamic optimization policies. Profiling data should be statistically representative of reality; otherwise, it could mislead Brainstorm’s optimization and result in reduced or even negative gain. As shown in Figure 24, the impact de-

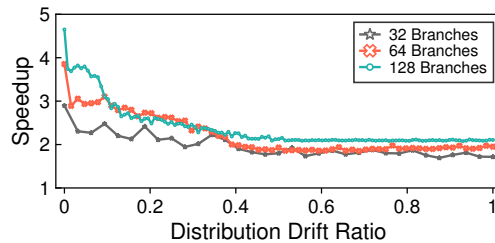


Figure 24: Speedup of Brainstorm’s dynamic horizontal fusion when the distribution of branch loads drifts from the statistics used for tuning GPU kernels.

pends on the models and the degree of drifts.

Figure 24 evaluate the impact of distribution drift on dynamic horizontal fusion. Based on the collected profiles, Brainstorm only tunes Conv2D kernels with 4 and 27 patches. Therefore, when a branch receives more than 4 patches, it needs to be padded to 27 patches running with the non-optimal 27-patch kernel. An initial dispatch of 4 patches per branch is made so that no padding is needed. To simulate increasing distribution drift, we add loads of some branches to 8 patches, which are less frequently appearing in the profile and thus not tuned by Brainstorm. We define the *distribution drift ratio* as the fraction of branches whose received patches differ from the tuned shapes (4 and 27 in this experiment). In Figure 24, we find the speedup of Brainstorm’s dynamic horizontal fusion BRT+HF diminishes with an increasing drift ratio, from  $4.65\times$  to  $2.11\times$ , compared with applying only vertical fusion. This is due to the wasted computation from the padding on branches receiving 8 patches.

The optimization policy needs to monitor profiles continuously collected by Brainstorm and triggers re-optimization when distribution drifts. It takes time for re-optimization (usually a few minutes), e.g., searching for a new placement, and tuning new GPU kernels. Therefore, during cold-start or re-optimization, the model execution does not use dynamic optimization. Currently, Brainstorm focuses on the mechanisms of enabling dynamic neural optimizations. We hope to inspire more advanced solutions to be robust to distribution drifts.

**More dynamic optimization opportunities.** Brainstorm can also be applied to training. When fine-tuning MoE-based Large Language Models, the statistics of expert activation can be leveraged similarly with inference, e.g., re-arranging the expert placements across GPUs to reduce communication volume. Moreover, many algorithms in Neural Architecture Search also design dynamic architectures (e.g., DARTS [49], SPOS [50]), whose activation is known only at runtime. Their latter stage of training may show more stable branch activation, which can be potentially exploited by Brainstorm.

To support training, there are still some engineering efforts that need to be resolved. Firstly, backward propagation



is needed for automatic differentiation in training, which is missed in the current implementation. Secondly, some operators may invalidate Brainstorm’s tracing for dynamic optimization. For instance, `Batchnorm` performs cross-*Cell* computing different from the *Cell*-level computation at inference, which requires manual specification.

Brainstorm can also be applied to dynamic sparsity, which uses different value/block-level sparsity patterns for different inputs (e.g., Longformer). To optimize their execution, Brainstorm needs to collect pattern statistics at a fine granularity. Then we can compile multiple specialized GPU kernels for different sparsity patterns (e.g., using SparTA [51]), and activate the most efficient one at runtime.

## 9 Related Works

**Deep Learning Frameworks for Dynamic NNs.** Popular DL frameworks can express dynamic neural networks via control-flow operators in static DFGs (e.g., TensorFlow 1.x [52]) or Python native control-flows (e.g., PyTorch [32], JAX [53], TensorFlow Eager [54]). They are capable of expressing dynamic neural networks in very flexible ways. However, their tensor-centric DFGs are hard to be analyzed at the sub-tensor level. As shown in §5.1, many dynamic NNs require *Cell*-level dataflow analysis, which the tensor-centric programming model misses. Brainstorm unifies how dynamic NN should be expressed so that the required information for dynamic optimization can be easily traced.

Optimization of dynamic NNs has also been studied in recent years, which mainly focuses on specific types of dynamism. Cava [55], DyNet [56], BatchMaker [57], TensorFlow Fold [58], DVABatch [59], ICE [60], and PAME [61] focus on dynamic batching [62] for the cases when the batch size is dynamic. Cortex [63] is a framework for recursive neural networks with compiler optimization. DietCode [64] is an auto-scheduler framework for optimizing dynamic shapes. Nimble [65] and DISC [66] are compilers to express and execute dynamic neural networks. Brainstorm is orthogonal to them by exploring a new optimization space that leverages runtime statistics of *Cell*-level dynamism.

**Optimization of deep neural networks.** Most optimizations of existing DL compilers and frameworks are proposed for optimizing static neural networks. TVM [39] expresses operators as loop optimization schedule primitives and search for efficient kernels. Anso [67] enlarges the search space via a hierarchical representation of the search space. Roller [68] uses a cost model to reduce the overhead of searching efficient kernels. XLA [47], Rammer [33], TASO [69], Tacker [70], TVM [39] also performs graph-level optimization on static DFGs, e.g., operator fusion. Pathways [71] proposes asynchronous distributed dataflow for large-scale distributed training. Brainstorm differs from these works in that it introduces

new optimization spaces for dynamic NNs through sub-tensor-level profiling. Brainstorm’s dynamic optimizations focus on exploring the runtime dynamism distribution of dynamic NNs, which are orthogonal to these works.

Moreover, Brainstorm’s *Router* separates the dynamic control flow from the dataflow graphs, which makes it easier to extract the static sub-networks for applying existing static optimizations. Brainstorm focuses on optimizing dynamic fragments in dynamic NNs and leaving optimizations of static sub-networks to existing compilers. With statistics of sub-tensor-level profiles, Brainstorm employs TVM [39] for kernel autotuning. Brainstorm can also leverage Pathways [71] to build an efficient execution plan to better fit the runtime dynamism, e.g., partition models with better affinity.

**Profile-guided optimization in modern programming languages.** Compilers for programming languages, e.g., HotSpot JVM [16], Dot-Net Core 2.0 [17], Clang [25], have supported dynamic optimization by collecting runtime statistics of programs and then compiling new optimized versions for future execution. Brainstorm is inspired by them and identifies new dynamic optimizations specific for dynamic NNs.

## 10 Conclusion

In this paper, we identify a new space of dynamic optimizations for dynamic NNs by collecting and analyzing runtime profiles to specialize the model execution to dynamism distribution. We propose Brainstorm, the first deep learning framework that optimizes the execution of dynamic NNs. The core of Brainstorm is *Cell* and *Router*, that lets model developers express dynamic NNs at the granularity of dynamism so that the necessary information for dynamic optimizations can be traced. Model developers can focus on designing the dynamic model architecture while leaving the optimization to the Brainstorm framework. In Brainstorm, we propose four dynamic optimizations leveraging the runtime profiles at different granularity. Our evaluation shows Brainstorm can accelerate popular dynamic neural networks by up to  $11.7\times$  ( $3.29\times$  on average) or reduces GPU memory usage by 42%.

## Acknowledgments

This work is partially sponsored by the National Natural Science Foundation of China (62232011, 62022057, 61832006), and Shanghai international science and technology collaboration project (21510713600). We thank the anonymous reviewers and our shepherd, Junfeng Yang, for their constructive feedback and suggestions. Zhenhua Han, Quan Chen, and Minyi Guo are the corresponding authors.

## References

- [1] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens Van Der Maaten, and Kilian Q Weinberger. Multi-scale dense networks for resource efficient image classification. *arXiv preprint arXiv:1703.09844*, 2017.
- [2] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. *Advances in neural information processing systems*, 30, 2017.
- [3] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.
- [4] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. *Advances in Neural Information Processing Systems*, 32, 2019.
- [5] Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11030–11039, 2020.
- [6] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.
- [7] Yue Guan, Zhengyi Li, Zhouhan Lin, Yuhao Zhu, Jingwen Leng, and Minyi Guo. Block-skim: Efficient question answering for transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10710–10719, 2022.
- [8] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, and Minyi Guo. Transkimmer: Transformer learns to layer-wise skim. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7275–7286, 2022.
- [9] Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2369–2378, 2020.
- [10] Xiangtao Kong, Hengyuan Zhao, Yu Qiao, and Chao Dong. Classsr: A general framework to accelerate super-resolution networks by data characteristic. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12016–12025, 2021.
- [11] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1039–1048, 2017.
- [12] Xiaoxiao Li, Ziwei Liu, Ping Luo, Chen Change Loy, and Xiaoou Tang. Not all pixels are equal: Difficulty-aware semantic segmentation via deep layer cascade. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3193–3202, 2017.
- [13] Yulin Wang, Kangchen Lv, Rui Huang, Shiji Song, Le Yang, and Gao Huang. Glance and focus: a dynamic approach to reducing spatial redundancy in image classification. *Advances in Neural Information Processing Systems*, 33:2432–2444, 2020.
- [14] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
- [15] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.
- [16] Java hotspot vm. <https://www.oracle.com/java/technologies/javase/javase-core-technologies-apis.html>. Accessed: 2022-11-11.
- [17] Profile-guided optimization in .net core 2.0. <https://devblogs.microsoft.com/dotnet/profile-guided-optimization-in-net-core-2-0/>. Accessed: 2022-11-11.
- [18] Mark Leone and R Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical report, Citeseer, 1997.
- [19] John Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, 2001.
- [20] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174, 2014.
- [21] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. Taming sparsely activated transformer with stochastic experts. *arXiv preprint arXiv:2110.04260*, 2021.

- [22] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *arXiv preprint arXiv:2206.03382*, 2022.
- [23] Amirhossein Habibian, Davide Abati, Taco S Cohen, and Babak Ehteshami Bejnordi. Skip-convolutions for efficient video processing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2695–2704, 2021.
- [24] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10):891–921, 1905.
- [25] Profile guided optimization in llvm. <https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization>. Accessed: 2022-11-11.
- [26] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.
- [27] Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference. *arXiv preprint arXiv:2110.03742*, 2021.
- [28] Yanwei Li, Lin Song, Yukang Chen, Zeming Li, Xiangyu Zhang, Xingang Wang, and Jian Sun. Learning dynamic routing for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8553–8562, 2020.
- [29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [30] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. Janus: fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.
- [31] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeongin Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. *Advances in Neural Information Processing Systems*, 34:1468–1480, 2021.
- [32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.
- [34] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. *Proceedings of Machine Learning and Systems*, 3:599–623, 2021.
- [35] Suzana Herculano-Houzel, Bruno Mota, Peiyan Wong, and Jon H Kaas. Connectivity-driven white matter scaling and folding in primate cerebral cortex. *Proceedings of the National Academy of Sciences*, 107(44):19008–19013, 2010.
- [36] Nvidia cuda: Unified memory programming. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd>. Accessed: 2022-11-11.
- [37] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.
- [38] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.
- [39] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
- [40] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-

- task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.
- [41] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.
- [42] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- [43] Milind Naphade, Shuo Wang, David C. Anastasiu, Zheng Tang, Ming-Ching Chang, Xiaodong Yang, Yue Yao, Liang Zheng, Pranamesh Chakraborty, Christian E. Lopez, Anuj Sharma, Qi Feng, Vitaly Ablavsky, and Stan Sclaroff. The 5th ai city challenge. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2021.
- [44] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.
- [45] Nvidia nccl. <https://developer.nvidia.com/nccl>. Accessed: 2022-11-11.
- [46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [47] Xla architecture. <https://www.tensorflow.org/xla/architecture>. Accessed: 2022-11-11.
- [48] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.
- [49] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XVI*, volume 12361 of *Lecture Notes in Computer Science*, pages 544–560. Springer, 2020.
- [50] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [51] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, Carlsbad, CA, July 2022. USENIX Association.
- [52] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [53] Jax: composable transformations of python+numpy programs. <http://github.com/google/jax>. Accessed: 2022-11-11.
- [54] Akshay Agrawal, Akshay Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *Proceedings of Machine Learning and Systems*, 1:178–189, 2019.
- [55] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, 2018.
- [56] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.
- [57] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.



- [58] Tensorflow fold. <https://github.com/tensorflow/fold>. Accessed: 2022-11-11.
- [59] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of {DNN} services on gpus. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.
- [60] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. Qos-aware irregular collaborative inference for improving throughput of dnn services. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 993–1006. IEEE Computer Society, 2022.
- [61] Shulai Zhang, Weihao Cui, Quan Chen, Zhengnian Zhang, Yue Guan, Jingwen Leng, Chao Li, and Minyi Guo. Pame: precision-aware multi-exit dnn serving for reducing latencies of batched inferences. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–12, 2022.
- [62] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.
- [63] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.
- [64] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems*, 4:848–863, 2022.
- [65] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.
- [66] Kai Zhu, WY Zhao, Zhen Zheng, TY Guo, PZ Zhao, JJ Bai, Jun Yang, XY Liu, LS Diao, and Wei Lin. Disc: A dynamic shape compiler for machine learning workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 89–95, 2021.
- [67] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Anso: Generating high-performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.
- [68] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. Roller: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.
- [69] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.
- [70] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 800–813. IEEE, 2022.
- [71] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ML. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

## A Artifact Appendix

### Abstract

Brainstorm unifies the programming of dynamic NNs with *Cell* and *Router* abstraction which enables a new space of dynamic optimizations for dynamic NNs. This artifact reproduces the main results of the evaluation in both single-GPU and multiple-GPU environments.

### Scope

This artifact will validate the following claims:

1. **Effectiveness of Brainstorm Abstraction:** By reproducing the experiments of [Figure 12](#), we can validate the effectiveness of Brainstorm's abstraction.
2. **Micro Benchmarks:** By reproducing the experiments of [Figures 13–15](#), we can validate the proposed dynamic optimizations with micro benchmarks.
3. **End-to-end Model Execution:** By reproducing the experiments of [Figures 16–23](#), we can validate the end-to-end latency of Brainstorm claimed in §7.

### Contents

In this artifact, we will reproduce the [Figures 12–23](#). Each figure has a shell script to reproduce and visualize the evaluation results automatically. In addition, we also provide a pre-built Docker image hosted on [Github Container Registry](#). Users can quickly initiate a container with this image, which has preconfigured experimental environments.

### Hosting

The artifact is hosted at <https://github.com/Raphael-Hao/brainstorm/tree/osdi2023ae>. To get the code, please git clone the Brainstorm repository and checkout to the `osdi2023ae` branch.

### Requirements

1. **Hardware Requirements:** [Figures 13, 15–17](#) and [21–23](#) requires a server with a NVIDIA A100 (80GB) GPU, [Figures 12, 14](#) and [18–20](#) requires a server with eight NVIDIA V100 GPUs.
2. **Software Requirements:** Please use docker to build the `docker/Dockerfile.update` to setup the environment for single and multiple-GPU experiments. A one-click script `python scripts/docker_gh_build.py -type latest` is also provided to build the image.
3. **CUDA Driver:** Larger than 11.3

## Tutorial

Please follow the instructions in `README.md` to reproduce the main results.



# AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models

Fan Lai<sup>2\*</sup>, Wei Zhang<sup>1</sup>, Rui Liu<sup>1</sup>, William Tsai<sup>1</sup>, Xiaohan Wei<sup>1</sup>, Yuxi Hu<sup>1</sup>, Sabin Devkota<sup>1</sup>, Jianyu Huang<sup>1</sup>, Jongsoo Park<sup>1</sup>, Xing Liu<sup>1</sup>, Zeliang Chen<sup>1</sup>, Ellie Wen<sup>1</sup>, Paul Rivera<sup>1</sup>, Jie You<sup>1</sup>, Chun-cheng Jason Chen<sup>1</sup>, Mosharaf Chowdhury<sup>2</sup>

<sup>1</sup>Meta    <sup>2</sup>University of Michigan

## Abstract

Deep learning recommendation models (DLRMs) are using increasingly larger embedding tables to represent categorical sparse features such as video genres. Each embedding row of the table represents the trainable weight vector for a specific instance of that feature. While increasing the number of embedding rows typically improves model accuracy by considering more feature instances, it can lead to larger deployment costs and slower model execution.

Unlike existing efforts that primarily focus on optimizing DLRMs for the given embedding, we present a complementary system, AdaEmbed, to reduce the size of embeddings needed for the same DLRM accuracy via in-training embedding pruning. Our key insight is that the access patterns and weights of different embeddings are heterogeneous across embedding rows, and dynamically change over the training process, implying varying embedding importance with respect to model accuracy. However, identifying important embeddings and then enforcing pruning for modern DLRMs with up to billions of embeddings (terabytes) is challenging. Given the total embedding size, AdaEmbed considers embeddings with higher runtime access frequencies and larger training gradients to be more important, and it dynamically prunes less important embeddings at scale to automatically determine per-feature embeddings. Our evaluations in industrial settings show that AdaEmbed saves 35-60% embedding size needed in deployment and improves model execution speed by 11-34%, while achieving noticeable accuracy gains.

## 1 Introduction

Deep learning recommendation models (DLRMs) are important to many online services, including Google advertisement display [9, 10], Netflix movie recommendations [15, 27], and Amazon e-commerce [40], and comprise up to 65% of AI cycles in Meta’s datacenters [13, 18]. Unlike conventional machine learning (ML) counterparts that train models on continuous input features (e.g., color values of images), DLRM inputs consist of continuous dense features (e.g., timestamp) and categorical sparse features (e.g., video genres). Each sparse feature is often associated with an embedding table, where each instance of that feature is represented by a trainable embedding row (weight vector). In the forward and backward

passes of model execution, the model reads and updates the embedding weights of accessed rows.

Because the accuracy of a DLRM typically increases with larger embeddings (e.g., by considering more feature instances), modern DLRM embedding size is ever growing (up to terabytes and billions of embeddings [13, 50]). This introduces multiple challenges. First, DLRMs often have stringent throughput and latency requirements for (online) training and inference [26, 45], but gigantic embeddings make computation [34], communication [4, 39] and memory optimizations [13, 52] challenging. To achieve desired model throughput, practical deployments often have to use hundreds of GPUs to hold embeddings [35]. Meanwhile, designing better embeddings (e.g., number of per-feature embedding rows and which embedding weights to retain) remains challenging because the exploration space increases with larger embeddings and requires intensive manual efforts [32, 49].

Unlike existing DLRM efforts that have primarily focused on optimizing the model’s execution speed for the given embeddings – e.g., by balancing embedding sharding [35, 52], accelerating embedding retrieval [39, 44], compressing embeddings [19, 48], or elastic resource scaling [45, 51] – we explore a complementary opportunity: *Can we fundamentally reduce the size of embedding needed for the same accuracy, by dynamically optimizing the per-feature embedding during model training?* Or, equivalently, *can we improve model accuracy for the given embedding size?* This is because unlike classic ML models, the DLRM model output (accuracy) is determined by the input data (e.g., accessed instances) and their embedding weights, and the input data is typically organized chronologically during training to account for the diverse and non-stationary user preferences [53]. Therefore, the access patterns and the weights of embeddings vary across embeddings rows and the training process (§2.2). This implies an opportunity to admit and prune embedding rows based on their heterogeneous importance to improve model accuracy.

In this paper, we introduce an *automated in-training pruning system* to Adaptively optimize per-feature Embeddings (AdaEmbed) for better model accuracy. For the given embedding size, AdaEmbed scalably identifies and retains embeddings that have larger importance to model accuracy at particular times during training. As a result, not only does it reduce human effort in embedding design, but it also cuts down the embedding size, thus the computational, network,

\*Work done while the author was working at Meta.



and memory resources, needed to achieve the same accuracy. AdaEmbed is complementary to and supports existing DLRM efforts with a few lines of code changes (§3).

Unfortunately, identifying important embeddings out of billions is non-trivial. To maximize the overall model accuracy, we should retain the embedding rows that affect model inputs more often (e.g., are frequently accessed) and that affect model outputs the most (e.g., have larger weights) (§4.1). However, the non-stationary data distribution during training leads to the spatiotemporal variation in the access frequency of different embeddings. e.g., new videos are posted and become popular, while some old ones lose popularity. Moreover, embedding weights change over training iterations and so does their impact. Once we prune an embedding’s weights from the GPU memory, we cannot accurately capture their importance to model accuracy as training moves on. Based on our analytical insights, embeddings with larger runtime gradients and higher access frequencies tend to accumulate larger embedding weights, and AdaEmbed prioritizes them when deciding which ones to retain. Moreover, we group features with similar feature-level characteristics (e.g., vector dimensions), and then identify important embeddings across feature groups to optimize the per-feature embedding size and which embedding to retain (§4.1).

Enforcing in-training pruning after identifying important embeddings is not straightforward either. Pruning for practical DLRMs can require reallocating millions of embedding rows and tens of gigabytes of embedding weights per training iteration, whereas each iteration takes only a few hundred milliseconds [4, 35]. While frequent pruning allows admitting important embeddings in a timely manner, thereby improving model accuracy, it can slow down model training by many hundred times (§4.2). To achieve a sweet spot between timely pruning and low overhead, AdaEmbed initiates pruning selectively when perceiving big changes in the importance distribution of all embeddings, thus reducing the number of pruning rounds needed while ensuring high accuracy. However, existing DLRM systems face difficulty in dynamically admitting and pruning embeddings at scale because they often rely on static and/or fixed-size embedding storage [1–3, 44]. AdaEmbed introduces a shim layer, Virtually Hashed Physically Indexed (VHPI) embedding, to support various embedding designs. VHPI decouples the management of embeddings from their physical weights, whereby it recycles the weight vector of embeddings to avoid intense memory allocation (§4.3).

We have implemented a system prototype of AdaEmbed (§5) and evaluated it using five industry models and months of data across hundreds of GPUs (§6). Our evaluations show that AdaEmbed can reduce 35-60% embedding size, implying comparable resource savings, and improve model execution speed by 11-34% without compromising model accuracy. Meanwhile, it achieves noticeable accuracy gains under the same embedding size, thus being able to reducing manual efforts by automatically finding better per-feature embeddings.

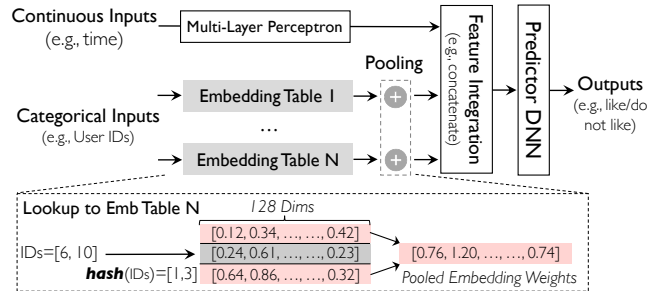


Figure 1: DLRM models consist of large embedding tables.

Overall, we make the following contributions in this paper:

1. We propose an in-training pruning system, AdaEmbed, to automatically optimize DLRM embeddings.
2. We introduce embedding importance to capture important embeddings and employ VHPI embedding to enforce scalable pruning, with few changes to existing designs.
3. We evaluate AdaEmbed in various real-world settings to show its resource savings and accuracy gains.

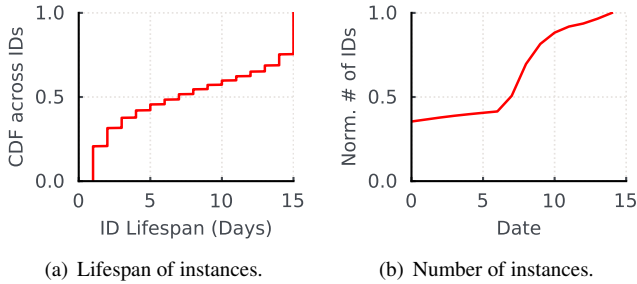
## 2 Background and Motivation

We start with a quick primer on DLRMs (§2.1), followed by the challenges it faces and inefficiencies of the state-of-the-art based on our analysis of real-world experiments (§2.2). Next, we highlight the opportunities that motivate our work (§2.3).

### 2.1 Deep Learning Recommendation Models

As shown in Figure 1, a DLRM consists of a combination of fully connected multiple-layer perceptrons (MLPs) to capture continuous dense features (e.g., timestamp), and a set of embedding tables to map various categorical sparse features (e.g., user and video IDs) to a dense representation. DLRMs can contain up to thousands of sparse features: each feature is typically associated with an embedding table, and each table can have millions of rows [15, 35, 52]. Each embedding row is a multi-dimensional weight vector (e.g., 128 floats) corresponding to a specific feature instance (e.g., a specific user ID of feature "User IDs").

DLRMs differ from traditional computer vision (CV) and natural language processing (NLP) models in that they require training on large volumes of data organized chronologically, to keep up with the latest recommendation trends. Hence, the distribution of training data changes over the training process. In the forward pass of model computation, each input sample includes a set of embedding IDs for each table to extract the corresponding embedding weights (vectors). To reduce the computation complexity, embedding weights of a sample will be pooled per table using the element-wise *pooling* operator, which typically takes the sum or maximum along each vector dimension (Figure 1). The pooled embedding weights of mini-batch samples are packed together with their intermediate outputs of dense features, forming a batch input to deeper layers. In the backward pass, the weights of the accessed embeddings are updated using the gradient.



**Figure 2: The number of sparse feature instances (IDs) increases rapidly over time, while the lifespan of instances is heterogeneous.**

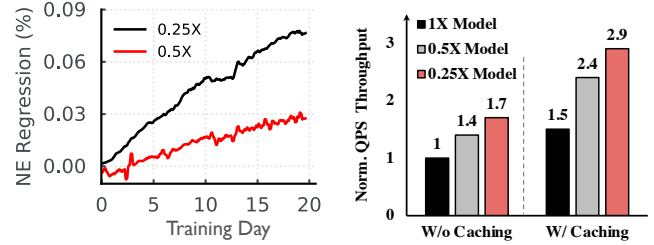
Due to the enormous number of sparse feature instances, their embedding weights can occupy more than 99% size of a commonly used model (up to several terabytes) [21]; so DL-RMs exhibit much larger memory intensity than conventional ML models (e.g., ResNet). As such, practical DLRM deployments use a combination of model parallelism for sparse feature layers and data parallelism for MLPs. The former allocates different embedding partitions across workers to avoid replicating them, and the latter enables concurrent processing of dense feature inputs [13]. Even so, model deployments often require hundreds of GPUs to achieve the desired model throughput (a few hundred milliseconds per iteration) [4, 35].

## 2.2 Challenges in DLRM Deployment

Due to its significant impact on revenue and numerous iterations needed to train a DLRM model, DLRM deployments follow the “achieve better accuracy and run as fast as possible” paradigm [35, 45, 52]. The execution speed and accuracy of a DLRM model are respectively measured by Query-Per-Second (QPS) throughput and Normalized Entropy (NE) loss [22]. Larger QPS and smaller NE indicate better performance, and any relative  $> 0.02\%$  NE gain is considered to be significant [13, 46]. However, optimizing both aspects leads to novel tussles and challenges in real-world deployments.

**Larger embedding sizes improve NE** Embedding size of modern DL-RMs is ever-growing to accommodate more embedding rows for sparse features and their instances [35, 44]. Figure 2 reports the size of the instance set over 15 days’ data in a real-world DLRM system. We observe that even though a small portion of the trained instances will seldom be accessed again in later days (Figure 2(a)), the total number of unique instances increases by  $1.5\times$  every week (Figure 2(b)). As DL-RMs are often trained on 1.5 months of data and retrained over time, the size of the instance set will eventually far exceed the embedding size. To cap the embedding size, existing designs often perform hashing on the raw instance IDs, and then use the hashed IDs to access their embedding rows [3].

Intuitively, using more embedding rows implies more instances are considered, thus enabling better data coverage for better NE. Figure 3(a) reports the impact of the embedding size on the NE regression at different times of training. NE regression denotes the accuracy degradation of using a smaller



(a) Large embeddings improve NE. (b) Large embeddings hurt QPS.

**Figure 3: Compared to the full ( $1\times$ ) model, smaller embedding sizes hurt model NE (i.e., larger NE regression), but improve QPS.  $0.25\times$  and  $0.5\times$  denote using 25% and 50% of the full model size.**

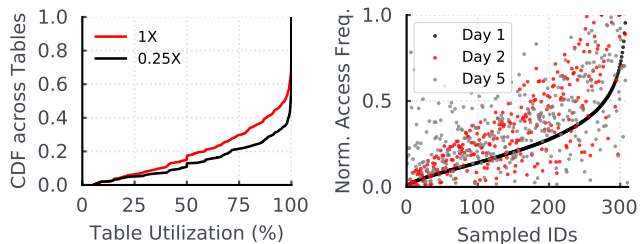
embedding size w.r.t. the full-size model. We notice that (i) using a smaller embedding size can greatly hurt NE. For example, reducing the number of embedding rows by 75% (i.e.,  $0.25\times$  model) results in  $\sim 0.02\%$  NE regression on Day 2; Worse, (ii) this NE regression inflates as the training evolves over time as more instances are spawned.

**Large embedding sizes hurt QPS** However, using more embeddings can slow down model execution and consume more machine resources in multiple execution phrases: (i) slower embedding access if we can not retain all embeddings in high-bandwidth GPUs; (ii) longer communication as we may need to transfer more embeddings over the network [4, 50]; and (iii) longer computation as more embeddings need to be computed on. Figure 3(b) shows, compared to the full model,  $0.5\times$  model achieves  $1.4\times$  QPS speedup in the same resource setting. Here, we note that state-of-the-art DLRM optimizations [35, 44], which cache and prefetch the embeddings to be accessed in future batches, cannot eliminate the QPS drop (Figure 3(b)). More importantly, they can be insufficient for online training and model serving as we may not know the input data in advance.

## 2.3 Opportunities for In-Training Pruning

For a given DLRM, recent advances have made considerable progress for efficient communication [4, 19, 39] and/or computation [13, 26, 35]. Instead, we focus on a complementary opportunity that *reduces the embedding size needed without NE regression, by adaptively pruning embeddings during model training*. Our approach is based on the following observations.

**Handcrafted embeddings are suboptimal** Designing optimal embeddings (e.g., deciding the number of per-feature embedding rows and which embedding weights to retain) is as yet an open problem in the ML community [14]. Hence, DLRM systems often decide the embedding size using human-defined rules, e.g., by estimating the feature popularity [14] and/or hyper-parameter tuning by model experts before training takes place [52]. Not only does this require great human effort and resources to explore, but it can also be suboptimal due to limited adaptivity at runtime (e.g., deciding which instance’s embedding to retain if many instances are generated).



(a) Utilization of embedding tables. (b) Heterogeneous ID access.

**Figure 4: Embedding access varies across IDs and over time, leading to distinct table utilization in existing embedding designs.**

Worse, existing DLRM systems often treat per-feature embedding tables individually for ease of management. This can underutilize or overload individual tables as data distribution changes over time. Indeed, when we analyze the table utilization in a one-day training window (i.e., number of accessed embeddings over the total number of embeddings on that day), we notice large heterogeneity (Figure 4(a)). Intuitively, tables that are fully utilized can degrade NE because many instances are hashed to the same embedding row, leading to hash collisions. However, underutilized tables cannot trade in their space during training because of the suboptimal pre-determined embedding size and inelastic embedding designs.

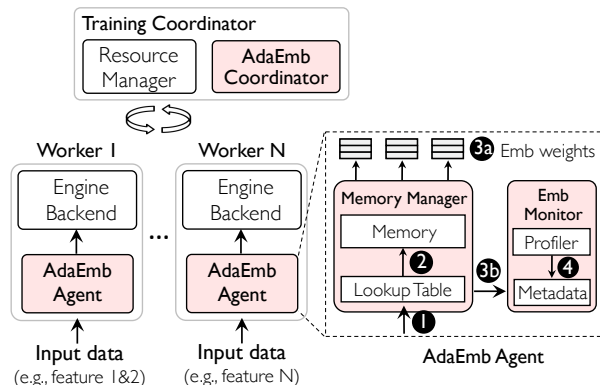
**Embeddings have heterogeneous characteristics** Figure 4(b) zooms into individual embedding rows, where the sampled IDs (i.e., x-axis) are ordered based on their access frequency on Day 1. We notice that the access frequency of embeddings varies across embedding IDs and over time, since user preferences change over time. We have similar observations on embedding weights too (§4.1). Since the model output (accuracy) is determined by the input instance (e.g., which embedding is accessed) and embedding weights, this implies a potential to identify and retain more important embeddings during training to maximize final model accuracy.

### 3 AdaEmbed Overview

In this paper, we introduce an *automated in-training pruning system*, AdaEmbed, to adaptively optimize per-feature embeddings at scale for better model accuracy. Unlike existing efforts for model pruning, which focus on conventional models [8, 11, 20] and/or prune model size when training completes [32], AdaEmbed automatically identifies and retains important embeddings for the given embedding size to improve performance while training is ongoing. Our evaluations in industrial settings show that in addition to saving resource throughout training, AdaEmbed provides superior model accuracy to its post-training pruning counterparts (§6.4).

AdaEmbed is a complementary system that acts as a shim layer atop today’s embedding designs (Figure 5). It has a central coordinator and a set of distributed on-worker agents:

- *AdaEmbed Coordinator*: It gathers the embedding information from agents, determines the global pruning decision, and orchestrates the agent to enforce the pruning.



**Figure 5: AdaEmbed overview and its in-training execution flow. AdaEmbed components are in red.**

- *Memory Manager*: It is located inside each AdaEmbed agent and manages the physical memory for today’s embedding designs. At runtime, it receives the pruning decision from the coordinator and executes pruning on local embedding weights.
- *Embedding Monitor*: It resides along with the memory manager to track embedding importance and reports the profiling results of the importance to the coordinator.

Figure 6 illustrates the interface of AdaEmbed, which supports existing DLRM systems in a few lines of code.

```

1 import AdaEmbed
2
3 def dlr_model_training():
4     # Wrap existing embedding modules
5     emb_agent = AdaEmbed.create_agent(
6         emb_tables=model.embs, pruning_config=config)
7
8     for _ in range(num_ iterations):
9         input_ids = get_next_data_batch()
10
11        # Look up physical embedding address
12        emb_physical_ids = emb_agent.look_up(input_ids)
13        feedback = model.train_step(emb_physical_ids)
14
15        # Update embedding importance with feedback
16        emb_agent.update_importance(input_ids, feedback)

```

**Figure 6: AdaEmbed supports existing DLRMs with minor changes.**

**Training Lifecycle** Similar to current DLRM deployments, ① each worker is in charge of a subset of sparse features, which is determined by the embedding partition of model parallelism. The worker processes the input data (i.e., a list of embedding IDs) of those features. ② However, the inputs are first forwarded to AdaEmbed agent to look up the physical address of each embedding’s weights (Line 12). ③ The physical address is then used to fetch the embedding weights for read and write operations. The rest of model training adheres to existing designs. ④ After each training iteration, the embedding monitor updates the embedding importance with the training feedback (Line 16). Periodically, it samples the importance of different embedding rows and notifies the coordinator of the profiling results. The coordinator determines how to prune embeddings subject to the total embedding size and guides the



memory manager to admit and prune embeddings at scale.

## 4 AdaEmbed Design

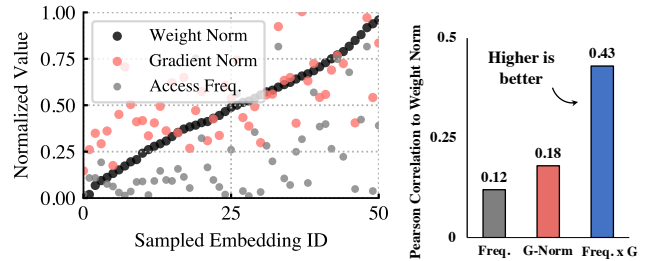
Practical DLRMs often contain hundreds of sparse features and up to billions of embedding rows [13, 50]. They run across hundreds of GPUs on non-stationary model inputs to get the desired model execution speed [4, 35]. These lead to the following challenges toward practical in-training pruning of embedding rows:

- *Heterogeneity*: The characteristics of embeddings (e.g., data distribution and embedding weights) vary across instances of the same feature. This, as well as the physical size of the embedding row, differs across features too. How to measure which embeddings are important to retain for better model accuracy (§4.1)?
- *Dynamics and Scalability*: The importance of individual embeddings varies over iterations at a sub-second speed. As such, improving model accuracy requires pruning in a timely manner to maximize the number of important embeddings. However, identifying important embeddings out of billions distributed across hundreds of workers, and then pruning on terabytes of embedding weights can lead to large overhead. How to orchestrate pruning under training dynamics (§4.2)? Additionally, how to efficiently enforce pruning on each worker’s memory to avoid throughput degradation (§4.3)?
- *Extensibility*: Existing systems are built atop a variety of embedding designs, such as key-value storage [44, 52] or highly optimized but fixed-size tensors [2, 3]. How to provide generic systems support to minimize modifications to existing DLRM systems (§4.3)?

### 4.1 Embedding Monitor: Identify Important Embeddings

Given the embedding size, we aim to trade the less important embedding rows for the more important ones. This requires us to consider the importance of each embedding row in terms of the contribution of its embedding weights to model accuracy, as well as its physical size. However, determining the optimal pruning strategy during training is challenging. First, the model output (accuracy) is affected by the complex interplay between input feature instances (e.g., which item IDs appear) and their embedding weights. Even with full model information after training completes, pruning is still a fundamental open problem in the ML literature [11, 32]. Second, during model training, this interplay becomes more intractable because of the large spatiotemporal variations in the distribution of model inputs and embedding weights (Figure 7(a)). Worse, once we prune an embedding’s weight vector, it is difficult to assess its impact on model accuracy as training moves on. These challenges are amplified by the need to account for feature-level heterogeneity too (e.g., different weight vector sizes across features).

AdaEmbed employs the embedding monitor to capture the



(a) Heterogeneous emb. characteristics. (b) Pearson analysis.

**Figure 7: (a) Embedding gradient and access frequency are heterogeneous, (b) while their combination reports a larger correlation to the embedding weights. A correlation value > 0.4 indicates a positive and medium to strong association.**

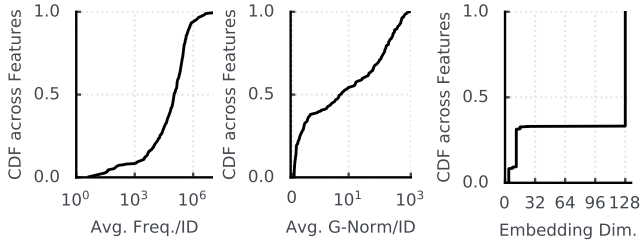
embedding importance of individual rows within the feature, and then extends it to identify important rows across features.

**Intra-Feature Embedding Importance** For embeddings of the same feature, we introduce a data- and model-aware importance metric  $EI(i)$  to capture the importance of each row  $i$  to model accuracy. Instead of relying on the embedding weights that become stale after being pruned,  $EI(i)$  is the runtime combination of access frequency and gradient, i.e.,  $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$ .  $\|\nabla g_t(i)\|$  is the L2-norm of  $i$ ’s gradient in iteration  $t$ , and  $freq_t(i)$  is the access frequency. So the embedding with a higher access frequency and a larger gradient norm is deemed more important. Here, collecting  $EI(i)$  introduces negligible overhead, because the embedding gradient is already generated during back-propagation of training regardless of AdaEmbed. Since the gradient is generated and shared by mini-batch samples [35], the importance of pruned-but-accessed embeddings will continue to be updated.

Our importance design is motivated by multiple factors:

- Intuitively, the output of sparse feature layers (i.e., pooled embedding weights) is often derived by taking the sum or maximum of input embedding weights (§2.1); so we should retain the embeddings that affect many model inputs (i.e., frequently accessed) and that affect model outputs more (i.e., larger weights). While we do not have information about future weights after pruning an embedding, we observe a strong correlation between our frequency-gradient combined metric and the final embedding weights when training converges (Figure 7). This is because frequent weight updates with large gradients typically result in larger weights.
- Theoretically, embedding rows are designed for training different bins of data instances: each bin holds only one type of category instance (i.e., a specific ID), and bins can have different data volumes (i.e., different access frequencies of IDs). Now, we want to select and retain certain bins (embeddings). This, in concept, is similar to the importance sampling problem in the ML literature [17, 25]: To improve model convergence by selecting the right bins to train the model, the optimal solution is to select bin  $i$  with a probability proportional to the aggregate gradients





(a) Heter. frequency. (b) Heter. grad. norms. (c) Heter. dimensions.

**Figure 8: Magnitudes of embedding access frequencies and gradients vary across features, making it hard to compare  $EI(i)$ .**

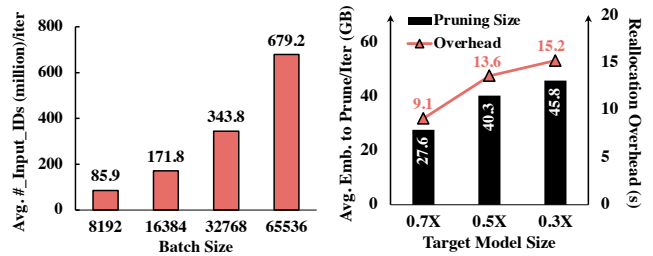
of training all that bin’s data. In our formulation, the training samples within the same bin are identical, because they correspond to the same specific ID. Therefore, the aggregate gradients herein is equivalent to the product of the number of training samples and the gradient of the individual sample (i.e.,  $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$ ).

Empirically, our fleet-wide evaluations show that our importance design outperforms its alternatives (§6.4).

Since the gradient and access frequency can fluctuate during training (e.g., due to the randomness in sampling mini-batches), we need to account for these uncertainties in  $EI(i)$ . Here, the embedding monitor considers  $EI(i)_t = freq_t(i) \times \|\nabla g_t(i)\| + EI(i)_{t-1}$ , whereby we reduce uncertainties in individual iterations and only need to update the importance of accessed embeddings. This is because the importance of not accessed embeddings remains unchanged as  $freq_t(i) = 0$ . In reality, only a subset of embeddings are accessed, so we can reduce the overhead significantly (§4.2). Moreover, to account for the temporal variation, we use a moving average that decays  $EI(i)$  by a factor of 0.8 every  $T$  iterations.

**Inter-Feature Group Pruning** Retaining important embeddings subject to the total size naturally leads to a global pruning design, in which we hope to allocate different embedding sizes to individual features. However, the values of embedding importance can vary across features by orders of magnitude. This can be due to features with fewer instances often having larger average access frequencies per embedding, and/or different initialization mechanisms of the embedding weights leading to gradients of different magnitudes (Figure 8). As such, directly using the intra-feature embedding importance for comparison across features can result in a large bias, as embeddings with greater importance values are not necessarily more important than those of other features. Moreover, as the dimension of embedding vectors of different features can vary (Figure 8(c)), deciding which embeddings are more valuable to retain becomes intricate when large embedding importance and vector size are in conflict.

Because we rely on the relative ranking of importance to determine pruning (e.g., prune the tail 40% less important embeddings), we can tackle the comparison bias across features using the popular normalization philosophy [16]; i.e., by normalizing each embedding’s importance by that



(a) Number of accessed IDs. (b) Size of weights to prune.

**Figure 9: (a) Each iteration accesses millions of embeddings. (b) Pruning needs to reallocate a large amount of embedding weights.**

feature’s distribution of all embeddings’ importance. This way, the embedding importance of different features takes on similar ranges of values, and the more important embeddings of each feature are still prioritized because of having larger relative importance values after normalization. The embedding monitor normalizes the embedding importance of each feature by the 95th percentile of its distribution (i.e.,  $EI(i)/EI_{95th}(feature(i))$ ) to avoid outliers.

Next, to account for different weight vector sizes across features, AdaEmbed groups features with the same embedding dimension and then performs global pruning within the feature group. In reality, DLRMs are configured with only a handful of distinct embedding dimensions (Figure 8(c)) to reduce hyper-parameter tuning and/or to achieve better parallelism (e.g., balancing embedding sharding [35, 52]). This implies a big opportunity to group many features, which already forms a large shared embedding size for inter-feature group pruning. By default, AdaEmbed initializes the per-group embedding size based on the number of in-group features and the total embedding size (i.e.,  $\frac{num\_group\_features \times group\_feature\_dim}{num\_features \times avg\_feature\_dim} \times total\_size$ ) to uniformly allocate the space to each dimension. Note that unused embedding storage will be picked up by other groups (§4.3). When developers have more advanced information about features (e.g., feature importance), AdaEmbed provides APIs for customizing feature groups and sizes (§5).

Our evaluations show that with importance normalization and group pruning, AdaEmbed achieves better resource savings and model accuracy (§6.3).

## 4.2 AdaEmbed Coordinator: Prune at Right Time

In real-world DLRM systems, each training iteration involves updating the importance of millions of embedding rows in terabyte-sized models (Figure 9(a)). At that scale, orchestrating hundreds of workers to prune leads to a trade-off between the pruning overhead and quality. Frequent pruning allows for better decision quality, i.e., maximizing the number of important embeddings all the time for potentially better model accuracy. Yet, pruning can require cleaning up and creating tens of gigabytes of embedding weights, which can take many seconds and significantly slow down the sub-second training iterations (Figure 9(b)). This trade-off becomes more

**Algorithm 1:** Pseudo-code of AdaEmbed runtime

```

1: weight_table ← EmbWeights()    ▷ Physical weight tables
2: emb_meta ← Init(weight_table)   ▷ VHPI metadata
3: pruning_start ← false          ▷ Enforce pruning or not
4: Function UpdateEmbs (input_ids, feedback) :
   /* Monitor: Update embedding importance
   asynchronously to model training. */
5:   UpdateImport(input_ids, feedback)
6:   if pruning_start == true then
7:     EnforcePruning()           ▷ Stall training
8:     pruning_start ← false
9: Function MonitorImportance (ProfilingInterval Δ) :
   /* Coordinator: asynchronously inspect big changes on
   the importance distribution via profiling across
   workers. */
10:  last_dist ← null
11:  while training == true do
12:    if mod(current_time, Δ) == 0 then
13:      cur_dist ← ProfileImportance()
14:      pruning_start ← Diff(last_dist, cur_dist) > p
15:      last_dist ← cur_dist
16: Function EnforcePruning () :
   /* Memory manager: Identify embedding rows to admit
   and prune subject to the given embedding size. */
17:  admit_emb, evict_emb ← IdentifyRecycleEmbs(
18:    emb_meta, weight_table.size)
   /* Redistribute the lookup mapping from the embedding
   ID to the weight vector, whereby admitted embedding
   rows can recycle the weight vector of pruned ones. */
19:  RedistLookup(emb_meta, admit_emb, evict_emb)
   /* Reset embedding weights for admitted embeddings. */
20:  weight_table.ResetEmbs(admit_emb)

```

intractable as a result of training dynamics; e.g., stochastic gradient descent can introduce large noise to embedding gradients, thus the embedding importance. As such, pruning too frequently can also be suboptimal (§6.4).

To find the sweet spot between pruning overhead and quality, AdaEmbed Coordinator decides the right time to prune to reduce the number of pruning rounds needed, and instructs the memory manager to minimize the overhead in each pruning round when pruning embedding weights (§4.3). Algorithm 1 outlines how AdaEmbed Coordinator orchestrates efficient embedding pruning. The embedding monitor updates the importance of accessed embeddings after each training iteration (Line 4), and periodically profiles embedding importance (Line 9). The results of the profiling will be sent to the coordinator. In the event of big changes in the importance distribution, the coordinator initiates a new pruning round and notifies the memory manager of the pruning decision (Line 9). The memory manager on each worker then executes pruning and admits new embedding weights at scale (Line 16).

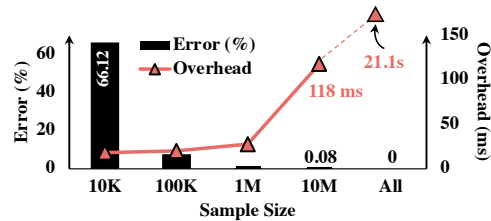


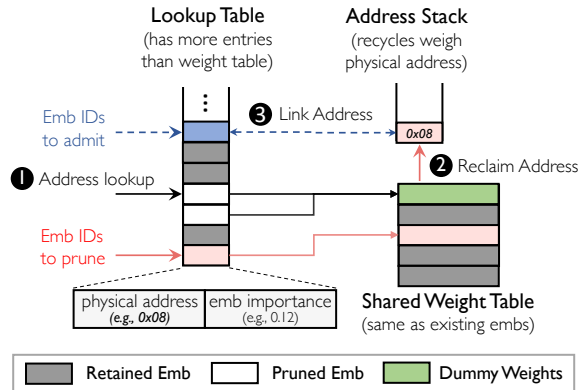
Figure 10: Profiling can get accurate results with little overhead.

Intuitively, pruning cares about the importance ranking of individual embeddings instead of their dynamic importance. Therefore, AdaEmbed coordinator relies on the importance distribution of all embeddings again, and initiates pruning if the importance distribution has changed greatly since the last pruning round. To effectively gather the importance distribution across hundreds of machines, each local agent samples a small portion,  $P$ , of embedding importance values on that agent. The coordinator then can estimate how many embeddings have crossed the pruning boundary, i.e., the number of embedding rows whose importance ranking has fallen below or risen above the  $X_{th}$  percentile of the distribution since the last pruning round.  $X_{th}$  is the cut-off importance boundary determined by the size limit (i.e.,  $\sum_{emb \in top\_X_{th}} size(emb) < total\_size$ ), and the agent will prune the weight vector of the embeddings whose importance value is smaller than the cut-off importance.

As shown in Figure 10, while more samples,  $P$ , allow for a more precise estimate of  $X_{th}$  importance, this will also increase the coordination overhead, such as in collecting importance distributed across hundreds of machines and then computing distribution changes. In fact, we can use the concentration theorem in the probability sampling [47] to decide the right number of samples.<sup>1</sup> This gives us  $\sim 5M$  embedding rows out of billions to sample on each machine, in order to ensure a deviation from the global ground truth of less than 1%. In addition to having a smaller computation overhead, this results in negligible network traffic,  $5M \times 4bytes \sim 20$  megabytes as  $EI(i)$  is a 4-byte float, over tens of Gbps network to the coordinator. As suggested by today’s data validation systems [7, 33], we consider a big change to have occurred and initiate pruning when more than  $c=5\%$  of the total embeddings cross the boundary (i.e., we need to prune and admit more than  $c\%$  embeddings), and issue this lightweight profiling per minute. This avoids the large overhead caused by pruning in each training iteration, while ensuring that the current embedding allocation is at most  $c\%$  worse than what we can achieve through pruning in each iteration. We show that profiling achieves a small deviation and little overhead (i.e., the 5M sample size in Figure 10).

**Convergence Analysis** As described in §4.1, our design of embedding importance draws inspiration from importance

<sup>1</sup>The minimum number of samples  $P$  needed to ensure  $Pr[|\bar{X} - E[\bar{X}]| < \epsilon] > \delta$  is  $P = \frac{(X_{max} - X_{min})^2 \ln(2/\delta)}{2\epsilon^2}$  for the distribution of variable  $X$ .  $E[\bar{X}]$ ,  $X_{max}$  and  $X_{min}$  are the expectation, maximum and minimum of  $X$ , respectively.



**Figure 11: VHPI employs lookup table to link each embedding to the weight vector, and recycles the vector of pruned embeddings without intense memory allocation.**

sampling, which has been shown in ML theory [17, 25, 31] for its ability to reduce gradient variance and accelerate training convergence. Empirically, our extensive evaluations using months of real-world data and models demonstrate that AdaEmbed consistently improves model accuracy by pruning at the right time, as opposed to pruning too frequently or infrequently (§6.4).

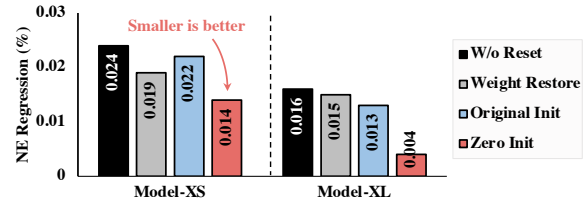
### 4.3 Memory Manager: Prune Weights at Scale

As the reallocation of embedding weights is hundreds of times slower than each training iteration (Figure 9(b)), reducing the number of pruning rounds needed is still far from achieving negligible overhead in practice (§6.2). To avoid intense memory reallocation, the memory manager of AdaEmbed employs a Virtually Hashed Physically Indexed (VHPI) design to decouple the management of embeddings from their physical weight vectors, whereby AdaEmbed can recycle the weight vectors of different embeddings to enable efficient pruning for a variety of existing embedding designs.

VHPI primarily consists of two parts (Figure 11):

- **Lookup table:** It stores the metadata information of each embedding instance, including the embedding importance (a `float32`), and the physical address (a `int64`) to that embedding’s weight vector. Compared to the weight vector, often a vector of 128 `float`, this payload information introduces a negligible memory footprint ( $\frac{3}{128} \sim 2\%$ ).
- **Weight table:** It is a monolithic physical table for embedding weight vectors. It remains the same as the embedding table of today’s DLRM systems, but it is shared across features under the orchestration of the memory manager.

Weights vectors of the pruned embeddings are not retained, while the metadata of all embeddings is always maintained in the lookup table. So the lookup table can include more entries (i.e., embedding IDs) than the weight table. This allows us to adaptively determine the link between embeddings and weight vectors to recycle weight vectors. Moreover, this can improve model accuracy by reducing hash collision (§6.4), as we can make the lookup table very large to accommodate



**Figure 12: Zero initialization performs better (0.5× model).**

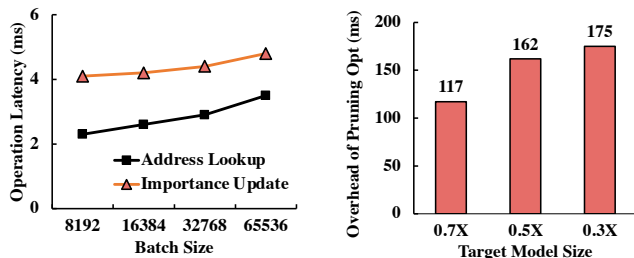
many embedding entries without expanding the weight table.

The memory manager performs two primitive operations for weight pruning at runtime (Figure 11):

- **Address lookup:** It looks up the physical weight address for each embedding ID to access its embedding weights. ① If that embedding row is pruned, to avoid breaking existing designs (e.g., missing weights due to pruning), lookup returns a shared physical address that points to a weights vector containing constant zeros. Access to this dummy vector will be folded on the execution backend due to the same entry, reducing redundant execution.
- **Weight allocation:** It executes the pruning decision to prune and admit embeddings. ② To prune an embedding row, VHPI first de-links and reclaims the current physical address of that embedding’s weight. It then sets the address of the pruned embedding’s lookup entry to the address of the shared dummy vector, redirecting the future access. ③ To admit an embedding, VHPI pops an available physical address and links this address with the lookup entry, thereby recycling the physical memory. Meanwhile, the memory manager resets the weight vector values to clean up the previously pruned weight state.

However, it is not straightforward to reset (i.e., reinitialize) the weight values for admitted embeddings, because the model herein is partially trained and the values of embedding weights already differ by orders of magnitude (Figure 7(a)). Improper initialization (e.g., random initialization) can introduce a large amount of noise to the retained embeddings. Eventually, this will hurt model accuracy, especially considering the noise from millions of admitted embeddings in each pruning round.

Here, we investigated four popular strategies to reset weight vectors (Figure 12): (1) *w/o reset*: inherit the weights of pruned embeddings without resetting them; (2) *weight restore*: evict previously pruned weights to extra storage (e.g., disk) and reinstate the weights when that embedding is reclaimed; (3) *original initialization*: randomly initialize embedding weights as at the start of training; and (4) *zero initialization*: reset embedding weights to zeros. Intuitively, the restored weights will become too stale since they were pruned (often thousands of iterations ago). Original initialization and w/o reset can introduce large noise, as the weights have already been of differing magnitudes. Here, we advocate resetting the weight vector values to zeros, as this can avoid large noise while allowing the admitted embedding to learn from scratch. Indeed, our real-world evaluations report that zero initialization outperforms its alternatives (Figure 12).



(a) Control-plane overhead. (b) Data-plane overhead.

**Figure 13: VHPI operations introduce little overhead.**

**Overhead Analysis** Among all the operations involved in VHPI, address lookup and importance update to the lookup table take place every iteration and consume a few milliseconds (Figure 13(a)). Weight pruning to the weight table consumes a few hundred milliseconds (Figure 13(b)), but it occurs every hundreds of iterations. Overall, these operations lead to little end-to-end overhead in large-scale deployments (§6.2).

## 5 Implementation

We implemented a system prototype of AdaEmbed to support distributed DLRM deployment across GPUs. Our implementation requires minor changes to existing DLRM systems.

**AdaEmbed Backend** AdaEmbed backend is implemented as GPU operators for fast execution. The VHPI metadata (e.g., embedding importance and weight address) are hosted on GPUs to process embeddings in parallel. The address lookup and importance update operations require no change to existing DLRM systems. As we need to reset the weight vector, the weight allocation operation requires existing frameworks to expose an API to access their weight table, but this requires a few lines of code change. The local agent interacts with the coordinator via TCP connections.

**Fault Tolerance** As a shim layer, AdaEmbed can be integrated into existing DLRM checkpoints by adding its state information to the model state. This not only minimizes the modification to existing designs, but also ensures that the saved AdaEmbed state conforms to the embedding weights at that time. When training is resumed, the model reloads the checkpoint, which restores the AdaEmbed state too. At runtime, AdaEmbed runs a lightweight daemon to back up VHPI metadata after each pruning round, and to resume its components if the current instances crash.

**Interfaces** AdaEmbed exposes Python APIs as the frontend (Figure 6), and it can also take *json* as input (Figure 14).

## 6 Evaluation

We evaluate AdaEmbed in real-world DLRM systems across hundreds of GPUs. Our evaluation results on different industrial models and months of data are summarized as follows:

- AdaEmbed can reduce 35-60% embedding size and improve model execution speed by 11-34% without compromising model accuracy (§6.2).

```

1  "adaembed_configs": {
2    "total_emb_size": "1 TB", // Total embedding size
3    "feature_configs": {
4      "default_group": {...},
5      "group_1": { // Features to use group pruning
6        "features": ["feature1", ...],
7        "total_emb_size": "200 GB",
8      } ... // Other feature groups
9    }
10 }

```

**Figure 14: Example embedding configuration in AdaEmbed.**

- AdaEmbed can reduce manual efforts by automatically finding better per-feature embeddings, achieving noticeable accuracy improvements (§6.2-§6.3);
- AdaEmbed improves performance over a wide range of settings and outperforms its design counterparts (§6.4);

### 6.1 Methodology

**Experimental setup** We use models and data from industry DLRM systems in the evaluation. Table 1 depicts high-level statistics of the model. They span different scales and recommendation tasks, including click-through rate prediction and ranking. We train each model on 14 days’ data to obtain the model *lifetime NE*, which indicates the cumulative model accuracy throughout training, and then test the model on the 15th day’s data to get the *evaluation NE*. Each day has many terabytes of data input.

The training batch size of each model is 65536, requiring tens of GPU nodes for the desired QPS. Each GPU node has 8 A100 GPUs with 40 GB of GPU memory. The GPUs are interconnected using 200 Gbps RoCE NICs.

**Baselines** To the best of our knowledge, AdaEmbed is the first system to support in-training embedding pruning, and is complementary to existing DLRM efforts. Our evaluations cover two primary baselines: (i) *w/o AdaEmbed*: an industry DLRM system without AdaEmbed support. Based on the access frequency of embedding rows in previous days, rows that are less frequently accessed are removed before training starts. This generates a pruned model derived from the full model; and (ii) different variants of AdaEmbed with changes in the pruning algorithm (§6.4). Here, we focus on the performance improvement of the *w/ AdaEmbed* setup, i.e., the setup using AdaEmbed.

**Metrics** We care about the (i) *memory saving* to achieve the same model accuracy as with the full model (i.e., without NE regression)<sup>2</sup>, because we want to minimize the embedding size for better model throughput and resource savings in deployment; (ii) *NE gain* that we can achieve using the same embedding size, since it not only minimizes manual efforts in configuring DLRM embeddings, but also implies higher revenues; and (iii) *overhead* that AdaEmbed introduces in model execution speed (i.e., QPS).

<sup>2</sup>A smaller Normalized Entropy (NE) loss indicates better model accuracy.



Model	# of Sparse Features (Approximate Value)	Raw Emb Size (Approximate Value)	# of GPUs	w/ Same Model NE		w/ Same Emb Size	
				Memory Saving	QPS Speedup	Avg. NE Gain (%)	QPS Overhead
Model-XS	1000	200 GB	32	≈ 35%	1.1×	0.015	0.4%
Model-S	600	350 GB	32	≈ 45%	1.2×	0.018	0.2%
Model-M	1000	1 TB	64	≈ 40%	1.2×	0.028	1.6%
Model-L	1000	1.1 TB	64	≈ 55%	1.3×	0.021	1.3%
Model-XL	800	1.5 TB	128	≈ 60%	1.3×	0.026	1.1%

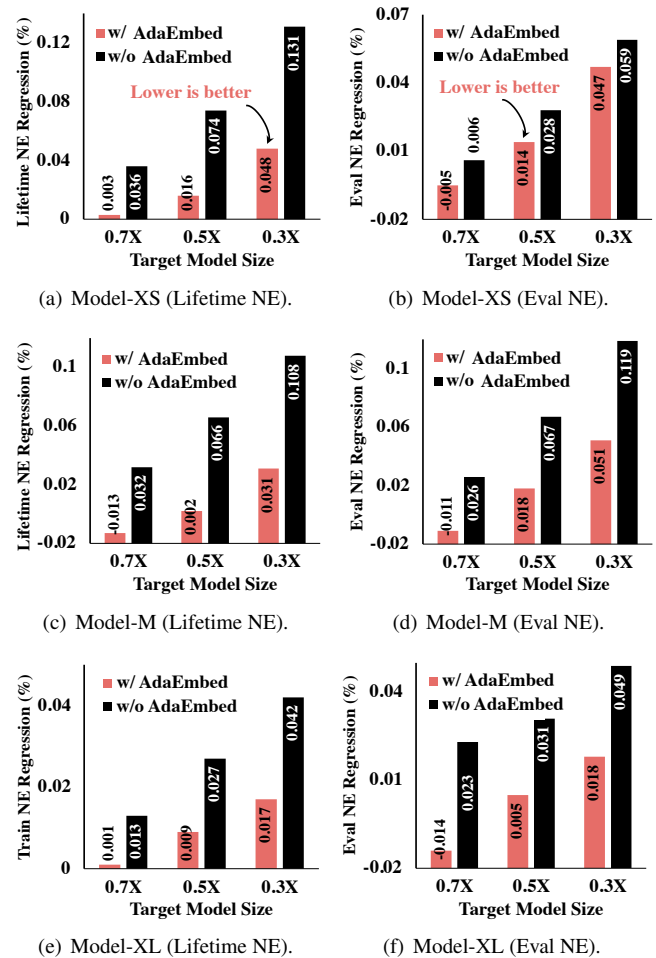
**Table 1: Summary of improvements. AdaEmbed reduces the embedding size needed for the same model accuracy (NE), while improving NE using the same embedding size. We report the approximate memory saving, since evaluating all memory settings is unaffordable.**

## 6.2 End-to-End Performance

Table 1 summarizes the key memory saving, NE gain, and overhead of five models at different scales. Meanwhile, Figure 16 zooms into three representative models and reports their performance under different target embedding sizes. In our evaluations, NE regression measures the accuracy loss w.r.t. the full model (i.e., 1× model), and any > 0.02% NE gap is considered to be significant [13, 30, 52].

**AdaEmbed cuts resource needs and improves QPS** We first evaluate how many embedding sizes we can reduce without sacrificing model NE. Yet, evaluating all embedding sizes to get accurate memory saving is unaffordable because training with each setup takes thousands of GPU hours. So, we enumerate 0.7× (i.e., cut the embedding size by 30%), 0.6×, 0.5×, 0.4×, and 0.3× of the full model size to approximate this embedding saving with no accuracy drop. Table 1 reports that (i) AdaEmbed reduces the model embedding size by 35-60% with no reduction in model accuracy. This implies that we can reduce the machine usage by nearly the same amount (e.g., using 50% fewer GPUs); (ii) the resource savings are more encouraging for large models (e.g., Model-XL vs. Model-XS). One reason behind this is that large models provide gigantic GPU memory for AdaEmbed to reallocate embeddings via inter-feature group pruning (§6.3); and (iii) alternatively, reducing the fundamental embedding size provides 1.1-1.3× faster model execution speed (i.e., QPS) when running the model on the same machines.

**AdaEmbed achieves better NE under the same size** Figure 16 illustrates that with AdaEmbed, models can achieve 0.011-0.077% better NE using the same embedding size. We notice that (i) AdaEmbed achieves consistently better NE across models and under different target embedding sizes than the baseline; (ii) we can achieve NE gains with smaller embedding sizes (e.g., 0.7× models) even when compared to the full model. This is because AdaEmbed can automatically learn better per-feature embeddings, like the size and which embeddings to retain. Meanwhile, pruning less important embeddings can reduce model overfitting, thereby improving model generalization (accuracy) [6]; and (iii) the lifetime NE



**Figure 16: AdaEmbed achieves better lifetime NE and evaluation NE. Better lifetime NE implies potentially better model accuracy for online learning deployment, while better evaluation NE indicates better accuracy after offline training (i.e., prior to launching online training). Both NEs are important metrics.**

gain is more prominent than that of the evaluation NE, because the former is closer to the online deployment (i.e., retraining on real-time data), where AdaEmbed is able to adapt to the latest data distribution.

**AdaEmbed introduces negligible overhead** As shown in Table 1, compared to the same-size model in the baseline,

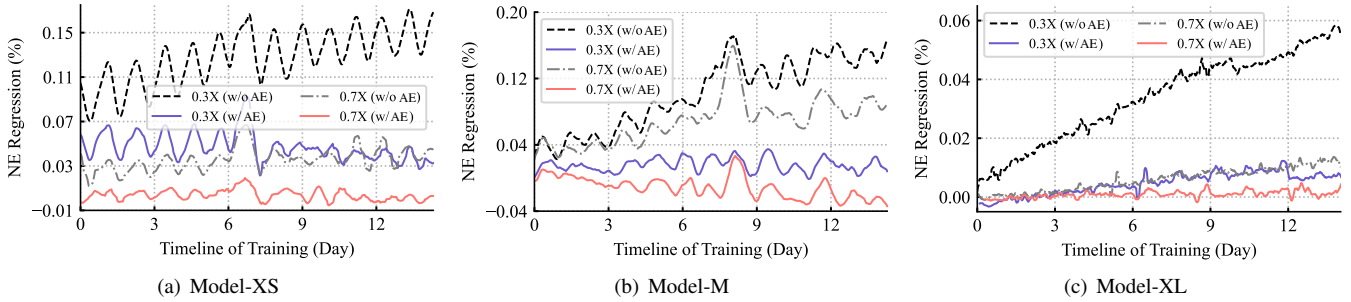


Figure 15: Models with AdaEmbed achieve consistently better NE over time. Troughs are due to data distribution shifting over days.

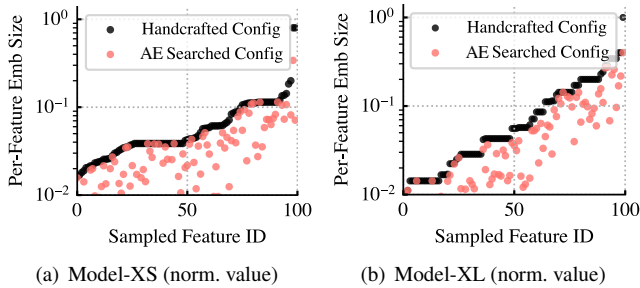


Figure 17: For the same NE w.r.t.  $1\times$  model, AdaEmbed learns better per-feature embedding configuration using smaller size.

AdaEmbed introduces negligible ( $< 2\%$ ) QPS overhead across scales of the deployment (e.g., from 32 to 128 GPUs and 200 GB to 1.5 TB models), because (i) AdaEmbed largely parallelizes operations (e.g., asynchronous importance update and multi-threading); (ii) coordinator selectively initiates pruning rounds; and (iii) the memory manager introduces VHPI to avoid intense reallocation of the physical weight. Note that the memory overhead is  $\sim 2\%$  as AdaEmbed introduces only two small buffers (i.e., the lookup address and embedding importance) in VHPI lookup table (§4.3).

### 6.3 Performance Breakdown

We next break down AdaEmbed performance by time, the characteristics of sparse features, and design components.

**Breakdown by Time** Figure 15 breaks down model NE by time, with each data point on the line representing the moving average of the NE over hourly data (i.e., window NE regression). The training encompasses 14 days of data. We observe that with AdaEmbed, we can achieve consistently small NE regression than the baseline over time.

Moreover, we notice that this NE regression exhibits diurnal variation (e.g., in Model-XS and Model-M). This is because the data distribution (e.g., user preference) of recommendation tasks can change drastically over days. As such, at the beginning of training on a new day’s data, the smaller model (e.g.,  $0.3\times$  model) will experience a larger NE regression as it has less space to accommodate new embedding IDs. However, as the model gradually adapts to the new distribution, this regression tones down. We note that AdaEmbed experiences less NE fluctuation due to its ability to identify

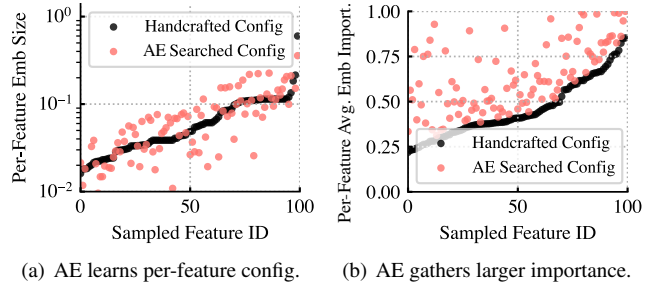


Figure 18: For the same size ( $0.5\times$  model), AdaEmbed retains more important embeddings to achieve better NE (Model-XS).

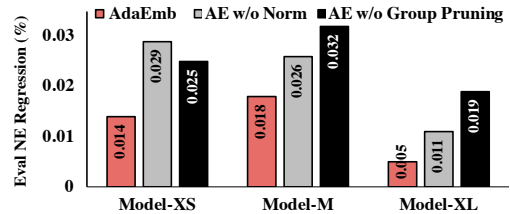
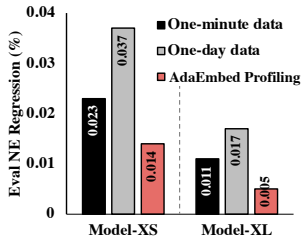


Figure 19: Performance breakdown of AdaEmbed (AE) design.

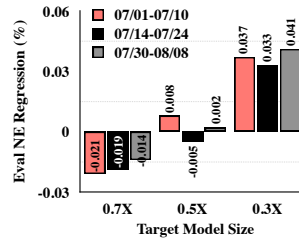
and retain important embeddings on the fly.

**Breakdown by Embedding Features** We next investigate whether AdaEmbed can reduce manual efforts by learning to use better embedding configurations. First, in achieving the same NE as the  $1\times$  model, AdaEmbed learns to use smaller embeddings for many features (Figure 17). Moreover, using the same embedding size w.r.t. the  $0.5\times$  model, AdaEmbed gathers larger average embedding importance on each feature than the handcrafted setup (Figure 18), implying that more important embeddings are retained under the same total size. More importantly, we notice that (i) our group pruning shares similar preferences to the handcrafted configuration. Specifically, AdaEmbed tends to allocate more embeddings to those features that the model expert also values highly. However, (ii) some features are allocated fewer embeddings but AdaEmbed eventually achieves better NE, indicating that AdaEmbed can automatically find better embedding configurations.

**Breakdown by Components** We break down our design into two variants (i) (AdaEmbed w/o Norm): disable importance normalization in group pruning; and (ii) (AdaEmbed



(a) Impact of pruning interval.



(b) Impact of dataset.

Figure 20: *AdaEmbed* achieves improvement across settings.

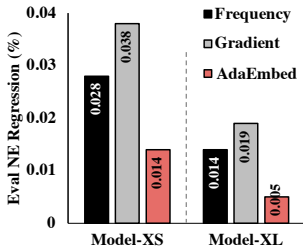


Figure 21: *AdaEmbed* outperforms importance alternatives.

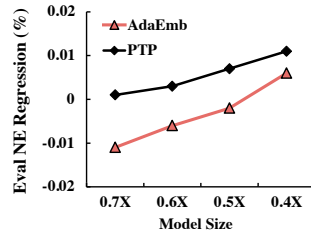


Figure 22: *AE* outperforms post-training pruning (PTP).

w/o Group): completely disable group pruning, so the per-feature embedding size is resized to X% of the full model. We notice that both normalization and group pruning contribute to better NE (Figure 19). This is because (i) group pruning allows greater flexibility to resize the per-feature embedding using the shared gigantic weight table; and (ii) importance normalization helps to reduce the inter-feature heterogeneity by prioritizing important embeddings of each feature when comparing embedding importance globally.

## 6.4 Sensitivity and Ablation Studies

**Impact of pruning frequency** *AdaEmbed* Coordinator initiates a pruning round when the importance distribution radically changes. Next, we evaluate the impact of pruning frequency by deterministically enforcing pruning after training every-minute ( $\sim 50$  training iterations) and every-day data ( $\sim 70K$  training iterations). Figure 20(a) reports that pruning too frequently and infrequently (i.e., pruning every one-minute and one-day data) both lead to suboptimal NE. The former is due to large training noise affecting instantaneous embedding importance, while the latter is due to *AdaEmbed* missing to admit important embeddings in a timely manner. Instead, the selective pruning of *AdaEmbed* achieves better performance by relying on the overall importance distribution at runtime.

**Impact of different data** Figure 20(b) reports the NE performance of model-S on three distinct datasets. Each training spans 10 days’ training data, and we report the evaluation NE on the data of day 11. While the NE gain varies slightly as the data distribution varies across dates, *AdaEmbed* consistently achieves 50% memory savings with no NE regression.

**Alternatives of embedding importance** We next experiment with different embedding importance designs in training 10 days’ data. Here, we consider using the frequency, gradient, and their combination (i.e., *AdaEmbed* design) as the embedding importance. We notice our frequency-gradient combination outperforms the alternatives. We note that this is consistent with the results of our Pearson analysis too, i.e., their combination has a stronger correlation to final embedding weights (Figure 8(b)). Instead, the access frequency and gradient only consider the data distribution and model characteristics, respectively, while DLRM accuracy depends on both aspects.

**In-training vs. post-training pruning** We compare *AdaEmbed* to its post-training pruning (PTP) counterpart like [20]. After model training is complete, PTP reduces the embedding size by pruning less important embeddings, as measured by our importance design. In fact, deploying PTP in real is often impractical (e.g., due to the need for online learning), and cannot achieve memory savings and/or QPS improvement during model training. Moreover, Figure 22 reports that *AdaEmbed* (i.e., in-training embedding pruning) can achieve better NE than PTP under the same embedding size, as the in-training design can adapt to the model performance at runtime and continuously optimize embeddings.

## 7 Related Work

**Deep Learning Recommendation Systems** Existing systems primarily focus on accelerating DLRM execution. NEO [18] co-optimizes embedding sharding and data parallelism. AIBox [52] and HierPS [51] overlap training execution on CPUs (using solid-state drives) and GPUs. Ekko [39] accelerates DLRM training over wide-area networks. TT-Rec [48] replaces embedding tables with matrix products to reduce memory footprints. Check-N-Run [13] reduces the bandwidth consumption for model checkpoints. Fleche [44] and Kraken [45] share the idea of sharing the weight table across features, but they focus on caching frequently accessed embeddings. *AdaEmbed* goes one step further by identifying the heterogeneous embedding importance to improve model accuracy during model training.

**Optimizations for Deep Learning** Recent ML advances have proposed various innovations for deep learning. TASO [23] and PET [41] perform tensor optimizations to improve model computation. Superneurons [42] and PipeSwitch [5] optimize instantaneous GPU memory by prefetching model layers based on their computation order. Similarly, ByteScheduler [37] and BytePS [24] accelerate the communication of distributed DNN training. Model-Keeper [28] warms up model training to reduce the amount of training execution needed. Egeria [43] adaptively freezes the training of model layers and bypasses their computation. These existing works focus primarily on conventional models, whereas DLRM models are often bottlenecked by memory-

intensive embeddings. Moreover, AdaEmbed is complementary to these efforts as AdaEmbed can further improve their optimized DLRM models.

**Model Pruning** Model pruning has been extensively studied to reduce model computation during training [11, 32], or to generate smaller models after training completes [8, 38]. Importance sampling [17, 29] performs weighted sampling on training data to achieve faster training convergence. Existing pruning systems and theories primarily focus on conventional CV and/or NLP counterparts by pruning only the dense layers [12, 20, 36]. However, in DLRMs, the gigantic embedding tables have become the bottleneck. This difference introduces novel challenges since the dense layers and embedding tables are distinct components with unique characteristics. For instance, dense layers are shared and accessed by all input samples, whereas each embedding row corresponds to a specific feature instance and is only accessed by it, leading to the heterogeneous importance of embeddings. Therefore, existing solutions are ill-suited for DLRMs.

## 8 Conclusion

This paper introduces AdaEmbed, an in-training embedding pruning system for better DLRM accuracy. AdaEmbed identifies embedding rows with larger importance to model accuracy, and then adaptively prunes less important embeddings to cap the total embedding size at scale. Our evaluations demonstrate that AdaEmbed can reduce manual efforts by automatically learning to use better per-feature embeddings, whereby it saves 35-60% embedding size needed in deployment, and achieves noticeable improvements on model accuracy and model execution speed.

## Acknowledgments

We thank our shepherd, Deepak Narayanan, and the anonymous reviewers for their insightful feedback that significantly improved the final paper. Further, we thank Hagay Lupesko and Jigar Desai for their continuous support throughout this project at Meta. This work was supported in part by NSF grants CNS-1909067, CNS-1900665, and CNS-2106184.

## References

- [1] HugeCTR: a high efficiency GPU framework designed for Click-Through-Rate (CTR) estimating training. <https://developer.nvidia.com/nvidia-merlin/hugectr>.
- [2] TensorFlow. <https://www.tensorflow.org/>.
- [3] TorchRec. <https://github.com/pytorch/torchrec>.
- [4] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. *arXiv preprint arXiv:2202.12429*, 2022.
- [5] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, November 2020.
- [6] Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. In *NeurIPS*, 2020.
- [7] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. Data validation for machine learning. In *SysML*, 2019.
- [8] Shih-Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. Directional pruning of deep neural networks. In *NeurIPS*, 2020.
- [9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS 2016*, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery.
- [10] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *RecSys*, 2016.
- [11] Xiaocong Du, Bhargav Bhushanam, Jiecao Yu, Dhruv Choudhary, Tianxiang Gao, Sherman Wong, Louis Feng, Jongsoo Park, Yu Cao, and Arun Kejariwal. Alternate model growth and pruning for efficient training of recommendation systems. In *arxiv.org/abs/2105.01064*, 2021.
- [12] Xiaocong Du, Bhargav Bhushanam, Jiecao Yu, Dhruv Choudhary, Tianxiang Gao, Sherman Wong, Louis Feng, Jongsoo Park, Yu Cao, and Arun Kejariwal. Alternate model growth and pruning for efficient training of recommendation systems. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021.
- [13] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *NSDI*, 2022.



- [14] A.A. Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. In *ISIT*, 2021.
- [15] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), December 2016.
- [16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. 2016.
- [17] Siddharth Gopal. Adaptive sampling for sgd by exploiting side information. In *ICML*, 2016.
- [18] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, and et al. The architectural implications of facebook’s dnn-based personalized recommendation. In *HPCA*, 2020.
- [19] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W. Mahoney. Training recommender systems at scale: Communication-efficient model and data parallelism. *KDD*, 2021.
- [20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015.
- [21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, and et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.
- [22] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD’14. Association for Computing Machinery, 2014.
- [23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.
- [24] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.
- [25] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.
- [26] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Young-jae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *ISCA*, 2020.
- [27] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.
- [28] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. ModelKeeper: Accelerating dnn training via automated training warmup. In *NSDI*, 2023.
- [29] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.
- [30] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: a hybrid system scaling deep learning based recommenders up to 100 trillion parameters. *arXiv preprint arXiv:2111.05897*, 2021.
- [31] Rui Liu, Tianyi Wu, and Barzan Mozafari. Adam with bandit sampling for deep learning. In *NeurIPS*, 2020.
- [32] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. Learnable embedding sizes for recommender systems. In *ICLR*, 2021.
- [33] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *MLSys*, 2022.
- [34] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, and et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *KDD*, 2021.
- [35] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair,

- Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. *ISCA*, 2022.
- [36] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *ASPLOS*, 2020.
- [37] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.
- [38] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning. In *NeurIPS*, 2020.
- [39] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In *OSDI*, 2022.
- [40] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. In *IEEE Internet Computing*, 2017.
- [41] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.
- [42] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *PPoPP*, 2018.
- [43] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *EuroSys*, 2023.
- [44] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient gpu embedding cache for personalized recommendations. *EuroSys*, 2022.
- [45] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In *SC*, 2020.
- [46] Li Yan, Choudhary Dhruv, Wei Xiaohan, Yuan Baichuan, Bhushanam Bhargav, Zhao Tuo, and Lan Guanghui. Frequency-aware sgd for efficient embedding learning with provable benefits. *ICLR*, 2022.
- [47] Ying Yan, Liang Jeff Chen, and Zheng Zhang. Error-bounded sampling for analytics on big sparse data. 2014.
- [48] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. TT-Rec: Tensor train compression for deep learning recommendation models. In *MLSys*, 2021.
- [49] Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. *NIPS'18*, 2018.
- [50] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: An FPGA-accelerated embedding-based retrieval system. In *OSDI*.
- [51] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *MLSys*, 2020.
- [52] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR prediction model training on a single node. *CIKM*, 2019.
- [53] Guorui Zhou, Chengru Song, Xiaoqiang Zhu, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *KDD*, 2018.



# BWoS: Formally Verified Block-based Work Stealing for Parallel Processing

Jiawei Wang<sup>1,2,3</sup>, Bohdan Trach<sup>1,2</sup>, Ming Fu<sup>1,2,\*</sup>, Diogo Behrens<sup>1,2</sup>, Jonathan Schwender<sup>1,2</sup>,  
Yutao Liu<sup>1,2</sup>, Jitang Lei<sup>1,2</sup>, Viktor Vafeiadis<sup>4</sup>, Hermann Härtig<sup>3</sup>, and Haibo Chen<sup>2,5</sup>

<sup>1</sup>Huawei Dresden Research Center    <sup>2</sup>Huawei Central Software Institute

<sup>3</sup>Technische Universität Dresden    <sup>4</sup>Max Planck Institute for Software Systems

<sup>5</sup>Shanghai Jiao Tong University

## Abstract

Work stealing is a widely-used scheduling technique for parallel processing on multicore. Each core owns a queue of tasks and avoids idling by stealing tasks from other queues. Prior work mostly focuses on balancing workload among cores, disregarding whether stealing may adversely impact the owner’s performance or hinder synchronization optimizations. Real-world industrial runtimes for parallel processing heavily rely on work-stealing queues for scalability, and such queues can become bottlenecks to their performance.

We present Block-based Work Stealing (BWoS), a novel and pragmatic design that splits per-core queues into multiple blocks. Thieves and owners rarely operate on the same blocks, greatly removing interferences and enabling aggressive optimizations on the owner’s synchronization with thieves. Furthermore, BWoS enables a novel probabilistic stealing policy that guarantees thieves steal from longer queues with higher probability. In our evaluation, using BWoS improves performance by up to 1.25x in the Renaissance macrobenchmark when applied to Java G1GC, provides an average 1.26x speedup in JSON processing when applied to Go runtime, and improves maximum throughput of Hyper HTTP server by 1.12x when applied to Rust Tokio runtime. In microbenchmarks, it provides 8-11x better performance than state-of-the-art designs. We have formally verified and optimized BWoS on weak memory models with a model-checking-based framework.

## 1 Introduction

Many language runtimes and similar systems (*e.g.*, JVM [104], Go [36], Rust’s Tokio [38]) divide their work into smaller units called *tasks*, which are executed asynchronously on multiple cores and whose execution can generate further tasks. To achieve good performance, the task scheduler has to ensure a

\*Ming Fu (ming.fu@huawei.com) is the corresponding author.

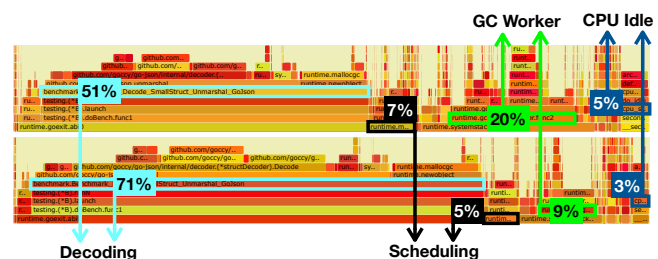


Figure 1: Profiling results for *go-json* complex object decoding ( $\sim 1\mu\text{s/op}$ ) benchmark [9], with the original work stealing queue (up) and with BWoS (down).

good *workload distribution* (preventing idle cores while there are pending tasks) with a low *scheduling overhead*.

Achieving these goals, however, is non-trivial. Storing the tasks in a single queue shared by all cores achieves optimal workload distribution, but incurs a huge overhead due to contention. Using per-core task queues minimizes the overhead per operation, but can easily lead to a skewed workload distribution, with some cores remaining idle while others have queued work.

*Work stealing* [51] is a trade-off between these two extremes: each core owns a queue (*owner*) and acts as both the *producer* and the *consumer* of its own queue to put and get tasks. When a core completes its tasks and the queue is empty, it then steals another task from the queue of another processing core to avoid idling (*thief*). A number of stealing policies [69, 76, 77, 83, 88, 100] have been proposed to choose the proper queue (*victim*) to steal from, which can bring significant speedups depending on the use-cases. Due to these features, work stealing is widely used in parallel computing [22, 35, 56, 64, 65, 85, 93, 97], parallel garbage collection [60, 68, 69, 96, 101], GPU environment [52, 54, 99, 102, 103], programming language runtimes [26, 36, 38, 50, 63, 80, 81], networking [86] and real-time systems [82].

However, as parallel processing is applied to more workloads, current implementations of work stealing become a bottleneck, especially for small tasks. For example, web frame-



works running over lightweight threading abstractions, such as Rust’s Tokio and Go’s goroutines, often contain many very small tasks, leading the Tokio runtime authors to observe that “the overhead from contending on the queue becomes significant” and even affects the end-to-end performance [28]. Similarly, high-performant garbage collectors, such as Java G1GC [13], rely on work stealing for parallelizing massive mark/sweep operations, which comprise only a few instructions. The work stealing overhead becomes a performance bottleneck for GC [68, 69, 96, 101].

As a third example, in Fig. 1, we profile the GoJson object decoding benchmark, which uses goroutines both for GC workers and for parsing complex objects. Only 51% of all CPU cycles constitute the useful workload (JSON decoding). The remaining cycles are spent on the runtime code, including 7% on lightweight thread scheduling, 20% on GC, 5% on kernel code idling the CPUs, *etc.* As both the scheduler and the GC code rely on work stealing, improving its performance can result in massive efficiency gains. Furthermore, the benefit is not limited to the above-mentioned scenarios, but expands to all fine-grained tasks parallel processing scenarios. Thus, we ask the following question: *How can we improve performance of work-stealing queues for fine-grained tasks to the benefit of a large range of common applications?*

Existing work-stealing queues suffer from four main sources of inefficiency:

**P1: Synchronization overhead.** Due to the possibility of a steal, local queues must use stronger atomic primitives (*e.g.*, atomic compare-and-swap and memory barriers) than a purely sequential queue. Queues with a FIFO policy are generally implemented as single-producer multiple-consumer (SPMC) queues [8, 17, 39, 47], thereby treating *steal* similarly to *get*, and thus distributing the costs of stealing equally between owner and thieves. This also applies to the existing block-based queues [106], which lack any optimizations specific to the work-stealing use case to achieve high performance in the presence of thieves (§6.2, §7). Queues with a LIFO policy, such as the well-known and widely-used ABP queue [35, 48, 104], suffer from memory barrier overhead [83, 98] to avoid the conflict between the owner and thieves, even when they operate on different tasks.

**P2: Thief-induced cache misses.** Since steals update the metadata shared between the owner and the thieves, they cause cache misses on subsequent accesses to the queue by its owner. This problem is especially apparent on unbalanced workloads, which feature high steal rates—for example, in the JVM Renaissance benchmarks [95], 10% of all items are stolen on average. Although strategies such as batching (*e.g.*, *steal-half* [66]) can reduce the frequency of steals, they often cause *overstealing* which introduces additional overhead (§2.1.3).

**P3: Victim selection.** To improve the workload distribution, advanced policies for selecting the victim queue to steal from require scanning the metadata of several queues, *e.g.*, to

find the longest queue. Doing so, however, causes contention for its owner and severely limits the improvement from advanced victim selection policies (§2.1.3).

**P4: Correctness under weak memory models (WMMs).** Correctly implementing concurrent work-stealing queues on weak memory architectures, such as Arm servers for datacenters [46, 70], is very challenging because it requires additional memory barriers to prevent unwanted reordering. Using redundant barriers can greatly reduce the performance of work-stealing [79], while not including enough barriers can lead to errors, such as in the C11 [6] version of the popular unbounded Chase-Lev deque translated from formally verified ARMv7 assembly [90]. Even the popular Rust Tokio runtime required a fix to its implementation of work stealing [2].

**Contribution.** In response, we introduce BWoS, a *block-based work stealing* (BWoS) bounded queue design, which provides a practical solution to these problems, drastically reducing the scheduling overhead of work stealing. Our solution is based on the following insights.

First, we split each per-core queue into multiple blocks with independent metadata and arrange for the owner and the thieves to synchronize at the block level. Therefore, in the common case where operations remain within a block, we can elide synchronization operations and achieve almost single-threaded performance (§3.2). Similarly, since a queue owner and the thieves share only block-local metadata, they do not interfere when operating on different blocks (§2.1). We can arrange for that to happen frequently by allowing stealing tasks from the middle of the queue.

Second, we improve victim selection with a *probabilistic policy*, which approximates selecting the longest queue (§3.4), while avoiding the severe interference typical of the prior state-of-the-art (§2.1), to which we can integrate NUMA-awareness and batching.

Finally, we ensure correctness under WMMs by verifying BWoS with the GenMC model checker [74, 75] and optimizing its choice of barriers with the VSync toolchain [92] (§5).

As a result, BWoS offers huge performance improvements over the state-of-the-art (§6). In microbenchmarks, BWoS achieves up to 8-11x throughput over other algorithms. In representative real-world macrobenchmarks, BWoS improves performance of Java industrial applications by up to 25% when applied to Java G1GC, increases throughput by 12.3% with 6.74% lower latency and 60.9% lower CPU utilization for Rust Hyper HTTP server when applied to the Tokio runtime, and speeds up JSON processing by 25.8% on average across 9 different libraries when applied to the Go runtime.

Returning to our motivating example (Fig. 1), applying BWoS to the Go runtime removes 29% of scheduling time, 55% of GC time, and 40% of CPU idle time, while increasing the CPU time ratio for useful work from 51% to 71%.

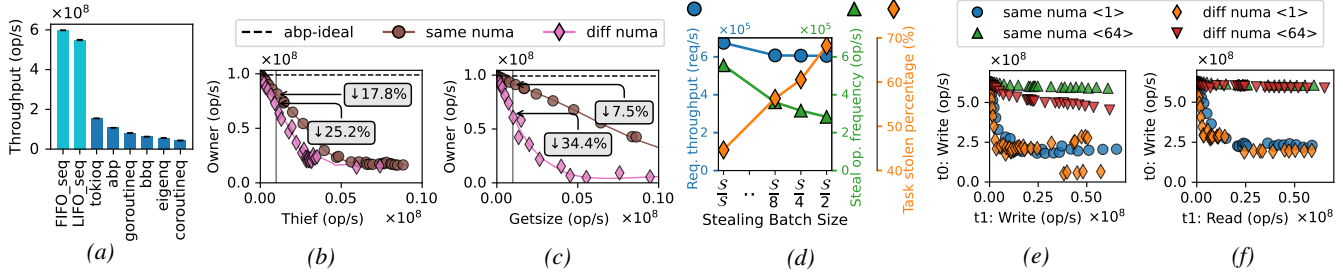


Figure 2: Motivating benchmarks: (a) Sequential performance of state-of-the-art work stealing algorithms. (b,c) Performance of the ABP queue owner depending on the frequency of (b) steal and (c) getsize operations. (d) Hyper HTTP server performance with different stealing batch sizes with the original Tokio work stealing queue:  $S$  is the victim queue size and  $S/2$  refers to the default steal half policy [66]. (e,f) Interference between two threads for two sizes of cacheline sets.

## 2 Background

**Task processing.** Tasks vary a lot among benchmarks. Their processing time ranges from a few nanoseconds (e.g., Java G1GC [13]), to microseconds (e.g., RPC [55, 73, 108]), and even to seconds (e.g., HPC tasks [35]). In this paper, we mainly focus on the nanosecond- and microsecond-scale tasks. Ignoring steals, tasks may be processed either:

- in FIFO (first-in-first-out) order, when minimizing processing latency is important (e.g., network connections), or
- in LIFO (last-in-first-out) order, when only the overall execution time matters, as is often the case with multithreaded fork-join programs [57].

We use the term *queue* to refer to the instances of work stealing data structures without implying a specific task ordering.

**Victim selection.** There are multiple policies for selecting the victim queue to steal from. *Random* [51] chooses one of the remaining queues uniformly at random: it has the least complexity but achieves poor load balancing. Size-based policies (e.g., *best of two* [88] and *best of many* [69]) scan the queues’ size to improve the load balance by stealing from a large queue. The NUMA-aware policy [77] was proposed to optimize the remote communication cost, by tending to steal from the queues in the local cache domain. Batch-based policies (e.g., *steal half* [66] is used in Go and Rust’s Tokio runtimes) allow thieves to steal multiple tasks at once to reduce their interference with the owner. Later in this section (§2.1.3), we will quantify these overhead sources to guide our queue design.

### 2.1 Performance Overhead Breakdown

Next, we analyze the state-of-the-art work stealing algorithms to dissect their performance issues, and motivate the design decisions of BWoS. Fig. 2 contains our experimental results on an x86 server [71].

#### 2.1.1 Cost of Synchronization Operations

As steals may happen at any time, strong atomic primitives are introduced for local queue manipulation. To quantify their cost, we first measure the throughput of the state-of-the-art

	Thief: cost of communication		Victim: cost of interference		Overhead reduction $1 - \frac{C_s + I_s}{C_d + I_d}$
	same node ( $C_s$ )	diff node ( $C_d$ )	same node ( $I_s$ )	diff node ( $I_d$ )	
abp	15ns	141ns	170ns	278ns	56%
ideal			–	–	90%

Table 1: Reducing the stealing overhead with a NUMA-aware policy.

work stealing algorithms on a sequential setup where an owner puts and gets data from its local queue, without any tasks ever being stolen (§6.2). We compare the results with the theoretical performance upper bound: a single-threaded FIFO (FIFO\_seq) or LIFO (LIFO\_seq) queue implementation [72] without support for steals. Although there is no owner-thief interference, these synchronization operations pose a huge overhead (Fig. 2a): throughput of these work stealing algorithms is less than 0.25x for FIFO-based (0.19x for LIFO-based) compared to the upper bound.

#### 2.1.2 Interference Cost with Thieves

To estimate how thieves affect the throughput of the owner, we consider an ABP queue benchmark with an owner and one thief, which steals tasks from the queue with various frequencies (one queue and two threads in total). As the “ideal” baseline, we take the single-threaded performance of the ABP queue (i.e., with no steals). To account for any NUMA effects in this measurement, we use two configurations, running the thief in the same or in different NUMA nodes.

As we can see in Fig. 2b, the thief significantly degrades the owner’s throughput: e.g., by stealing only 1% of the tasks, the owner’s throughput drops by 17.8% when the thief is in the same NUMA node, and by 25.2% when it runs in a different NUMA node. This degradation happens because of the cache interference between the owner and the thief on the shared metadata. We will further explain this in §2.2.

#### 2.1.3 Overhead due to Victim Selection

There are two main sources of stealing overhead: first, a sub-optimal victim selection can lead to workload imbalance trig-

gering more stealing; second, the cost of steal operations.

**Size-based policies.** Policies like *best of two* [88] or *best of many* [69] read global metadata of multiple queues (their length) to determine the victim. Somewhat surprisingly, as shown in Fig. 2c, these reads introduce significant overhead for the owner, especially in the cross-NUMA scenario: even with a *getsize* frequency of only 1%, the owner throughput drops by 34.4%. This is further amplified as *getsize* is called multiple times for a single steal.

Therefore, for size-based policies, although reading more queues' sizes (e.g., *best of many* [69]) can achieve better load balance, it inevitably induces more slowdown to the owners of these queues (§6).

**NUMA-aware policies.** NUMA-aware policies [77] try to reduce the overhead of each steal by prioritizing the stealing from queues in the same NUMA node. We observe that although such NUMA-aware policies can reduce the overhead of steals by 56% in the case of our ABP queue benchmark, they fail to achieve their full potential.

In Table 1, we break down the overhead of stealing in the ABP queue into its two main parts: the thief's communication cost and the owner's interference penalty. The former is 141ns when the thief and owner run on different NUMA nodes (measured by Intel MCA [21]), and reduces to 15ns (consistent with the L3 cache access latency [7]) when they are at the same NUMA node. The victim's interference penalty is 170ns and 278ns for cases of thief and victim running on the same ( $I_s$ ) and different ( $I_d$ ) NUMA domains respectively. NUMA-aware policies with existing queues can typically eliminate the first communication overhead, while leaving the second interference overhead not sufficiently optimized.

With long enough queues, steals could ideally happen at a different part of the queue and cause no interference to the victim. This would reduce  $I_s$  and  $I_d$  to zero, resulting in a 90% improvement due to NUMA-awareness (rather than 56%).

**Batch-based policies.** Batch-based policies steal more tasks at once with the aim of reducing the frequency of steals. Indeed, in the Hyper HTTP server benchmark (see Fig. 2d), choosing larger batch sizes leads to a reduction in the number of steal operations. These larger steal operations, however, make the workload even less balanced (i.e. percentage of stolen tasks increases), which results in additional overhead (e.g., task ping-pong), canceling out the overhead reduction due to the fewer steals: the end-to-end throughput remains roughly the same.

## 2.2 Recap to Motivate BWoS

In summary, the owner's performance suffers both from the synchronization cost, and the interference with thieves (due to victim selection and task stealing). This interference occurs because of cache contention on the queue metadata: write-write interference with *steal*, and read-write interference with *getsize* in size-based stealing policies.

To better understand the effects of these types of cache contention, we conduct a simple microbenchmark with two threads: thread  $t_0$  continuously writes to a cacheline, while thread  $t_1$  either reads or writes to a cacheline with a specified frequency (Figs. 2e and 2f). The cachelines for  $t_0$  and  $t_1$  are independently and randomly chosen on each iteration out of the cacheline sets of two sizes: 1 or 64.

In both cases, the cache contention on a single cacheline significantly harms the throughput of  $t_0$ , regardless of the NUMA domain proximity. Introducing multiple cachelines (64 in this case) reduces the contention and significantly improves the throughput. Therefore, in the design of BWoS we separate the metadata.

## 3 Design

BWoS is based on a conceptually simple idea: the queue's storage is split into a number of blocks, and the global mutable metadata shared between thieves and owner is replaced with the per-block instances.

The structure of BWoS queue facilitates abstracting the operations into *block advancement* that works across blocks, and *fast path* that operates inside of the block chosen by the block advancement (§3.1). Moving most of the synchronization from the fast path to the block advancement allows BWoS to fully reap the performance benefit indicated by our previous observation (§2.1.1) thus approaching the theoretical upper bound. *get* and *steal* always happen on different blocks. We carefully construct the algorithm such that thieves cannot obstruct the progress of *get*, while *get* can safely *takeover* a block from thieves operating on it without waiting for them. For complexity consideration, we don't prohibit *put* and *steal* in the same block<sup>1</sup>, as they can synchronize with the weak barriers without losing performance (§6.2).

As metadata is also split per block, thieves and the owner are likely to operate on different blocks and thus update different metadata. As explained in §2.2, this reduces the interference between thieves and the owner. For FIFO-based BWoS, block-local metadata allows stealing from the middle of the queue, without enforcing the SPMC queue restriction of always stealing the oldest task, which is not required by the workloads.

BWoS can benefit from NUMA-aware policies more than other queues because the reduction in interference for the victim makes both constituents of cross-NUMA-domain stealing overhead negligible (Table 1). Furthermore, unlike batch-based policies, stealing policies integrated with BWoS can focus on balancing the workload itself without worrying about the interference from frequently called *steal*.

<sup>1</sup>Nevertheless, it is guaranteed automatically in LIFO BWoS.



```

1  bool queue<E>::put(E e){
2  again:
3    blk = blk_to_put();
4    switch(blk.put(e))
5    case success():
6      return true;
7    case blk_done(rnd):
8      if (adv_blk_put(blk, rnd))
9        goto again;
10     else return false;
11  }
12  E queue<E>::get(){
13  again:
14    blk = blk_to_get();
15    switch(blk.get())
16    case success(e):
17      return e;
18    case empty:
19      return null;
20    case blk_done(rnd):
21      if (adv_blk_get(blk, rnd))
22        goto again;
23     else return null;
24  }
25  E queue<E>::steal(){
26  again:
27    v, blk = blk_to_steal();
28    switch(blk.steal())
29    case success(e):
30      return e;
31    case empty:
32      return null;
33    case conflict:
34      goto again;
35    case blk_done(rnd):
36      if (adv_blk_steal(v,
37        blk, rnd))
38        goto again;
39     else return null;
40  }

```

Figure 3: Pseudocode of put, get, and steal operations.

### 3.1 Bird’s-Eye View of the Queue

To better understand the block-based approach, let’s consider the *put*, *get*, and *steal* operations of the BWoS queue (Fig. 3). For each of these operations, the first step is to select a block to work on (lines 3, 14, and 27). The owner uses the *top* block for put and get for the LIFO BWoS, and gets from the *front* block and puts to the *back* block for the FIFO BWoS. In this case, top, back, and front block pointers are owner-exclusive metadata which is unavailable to the thieves. For *steal*, the choice of the block is more complicated and we will explain it in a later section (§3.4).

After selecting the block, operations execute the fast path (lines 4, 15, and 28), which may return one of the three results: (1) The fast path succeeds, returning the value for *get* and *steal*. (2) The fast path fails because there is no data to consume (lines 18 and 31) or because a thief detects a conflict with other thieves or with the owner due to the takeover (line 33). In case of a conflict, the fast path is retried (line 34), otherwise null value is returned. (3) The *margin* (beginning or end) of the current block is reached (lines 7, 20, and 35). In this case, the operation tries to move to the next block by performing the block advancement, and retries if it succeeds, otherwise returns the empty or full queue status.

Splitting the global metadata into block-level instances enables splitting the operations into the fast path and block advancement, which increases the performance by keeping the fast path extremely lightweight. However, the lack of global mutable metadata shared between owner and thieves raises additional challenges, which are mostly delegated to the block advancement—it is now responsible for maintaining complex block-level invariants. We introduce the following invariants:

- (1) *put* never overwrites unconsumed data;
- (2) *steal* and *get* never read the same data;
- (3) *steal* and *get* never read data that has been read before;
- (4) *steal* in progress cannot prevent *get* from reading from a thieves’ block. Before explaining fast path and block ad-

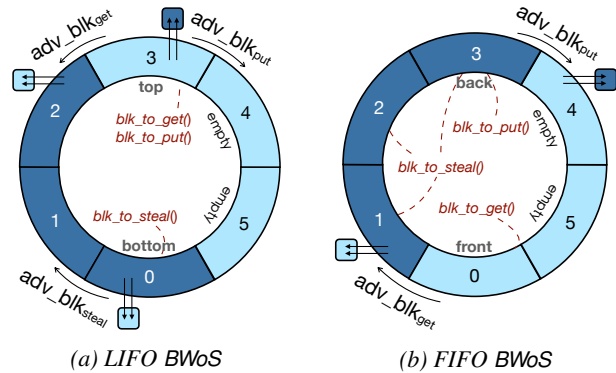


Figure 4: Block-level synchronization in BWoS.

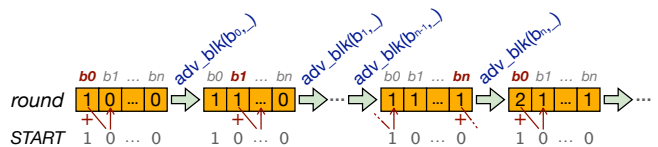


Figure 5: Update of round numbers in each block.

vancement implementations, we introduce two key concepts we rely on to ensure that the abovementioned invariants hold: *block-level synchronization* (§3.2) and *round control* (§3.3).

### 3.2 Block-level Synchronization

Block-level synchronization is the key responsibility of the block advancement and ensures that thieves never steal from the block currently used for get operations. Each block is owned either by the owner or by the thieves. For example, in Fig. 4, blocks with lighter and darker colors belong to the owner and thieves respectively. The owner grants a block to the thieves, or takes a block back from them with block advancement. More specifically, for LIFO BWoS, *get* advances to the *preceding* block (3 to 2) and takes it over from thieves; *put* grants the current one and advances to the *following* block (3 to 4). For FIFO BWoS, *get* (resp. *put*) advances and takes over (resp. grants) the following block.

The grant and takeover procedures are based on the *thief index*—an entry in the block metadata that indicates the stealing location inside the block. Takeover sets this index to the block margin with an atomic exchange, and uses the old value as the threshold between the owner and the ongoing thieves in this block. This ensures that owner is not blocked by thieves when it takes over the block. Moreover, concurrent owner and thieves never read the same data because the threshold between them is set atomically. Similarly, the grant procedure transfers the block to thieves by writing the threshold to the thief index. We will introduce the details in §4.2.1.

### 3.3 Round Control

Each block also records *round numbers* of the last data access. When advancing block, the current block’s round is copied over to the next block; except in the case of a wrap-around,



where the block number is increased by 1 (Fig. 5).

In fact, there are producer, consumer, and thief round numbers in each block. When the producer tries to write round  $r$ 's data into a block, the consumer and thieves must have finished reading all data with round  $r - 1$  from that block; so that the producer never overwrites any unread data. Similarly, when the consumer or a thief tries to read round  $r$ 's data from a block, the producer's round at that block must already be  $r$ ; this prevents reading any data twice, or reading data that was never written. Details can be found in §4.2.2.

### 3.4 Probabilistic Stealing

As discussed in §2.1.3, size-based policies can achieve better load balance at the cost of degrading the performance of the owner of each queue. Calculating the size is even harder in our setting because the appropriate metadata is distributed across all blocks. However, BWoS brings an opportunity to have a new size-based, probabilistic stealing policy, which can provide strong load balance without adversely affecting the owner's performance.

We ensure strong load balancing by making the probability of choosing a queue as a victim proportional to its size. We implement this approach with a two-phase algorithm: the  $P_{select}$  phase first selects a potential victim randomly, and then the  $P_{accept}$  phase decides whether to steal from it with probability  $S/C$ , where  $S$  is the selected queue's size and  $C$  is its capacity; otherwise (with probability  $1 - S/C$ ) it returns to  $P_{select}$  for a new iteration.

Therefore, given a pool of  $N$  queues each with the same capacity and a selector in  $P_{select}$  that selects each queue with equal probability,  $P_{accept}$  can guarantee that the probability of a thief stealing from a queue is proportional to its size.

To minimize the impact on the owner's performance, instead of measuring  $S$ , we estimate  $S/C$  directly by sampling. The thief chooses a random block from *all* blocks of the queue and checks if it has data *available for stealing*, where the probability of returning true is close to  $S/C$ . As the thief reads only one block's metadata, its interference with the owner is minimal (cf. §2.2).

For FIFO BWoS, the above approach can achieve zero-overhead for steals: after the estimation returns true, we can steal from the block used for estimation directly, as block-local metadata enables thieves to steal from any block which has been granted to thieves. We call this instance of applying our probabilistic stealing policy to FIFO BWoS a randomized stealing procedure.

For LIFO BWoS, stealing still happens from the *bottom* block (Fig. 4). Thieves advance to the following block when they finish the current one. For FIFO BWoS, thieves do not advance block when randomized stealing is enabled, and fall back to the stealing policy for selection of the new queue and block instead (§3.4). In this case, the operation to advance to the next block on stealing (Fig. 3 line 36) becomes a no-op.

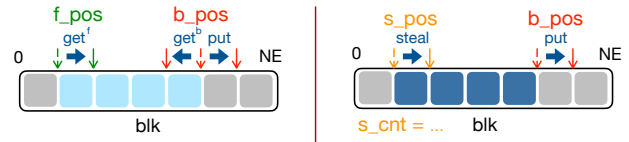


Figure 6: Put, get, and steal operations inside the block.

Moreover, we can further combine the probabilistic stealing policy with a variety of selectors for  $P_{select}$  phase (e.g., from NUMA-aware policy), to benefit from both better workload balance and reduced stealing cost. Results show that the hybrid probabilistic NUMA-aware policy brings the best performance to BWoS (§6).

## 4 Implementation

### 4.1 Single-Block Operations (Fast Path)

Let's consider how *put*, *get*, and *steal* operations inside the block are implemented (lines 4, 15, and 28 in Fig. 3). Because *get* and *steal* always happen on different blocks, we only need to consider two cases of multiple operations in a block: producer-consumer and producer-thieves (Fig 6).

To support these cases, each block has 4 metadata variables: entries which are ready for the consumer in the block are between the *front position* ( $f\_pos$ ) and *back position* ( $b\_pos$ ), while thieves use the *stealing position* ( $s\_pos$ ) and a counter of *finished steals* in the block ( $s\_cnt$ ) for coordinating among themselves and with the producer respectively.

To produce a value, *put* first checks whether it reaches the block margin NE (number of entries), if not, writes the data into the producer position ( $b\_pos$ ), and lets it point to the next entry.

To consume a value, there are two *get* operations,  $get^f$  and  $get^b$ , which correspond to the FIFO and LIFO BWoS respectively. *get* checks whether the block margin has been reached, or if the block has run out of data ( $f\_pos$  has reached  $b\_pos$ ), if not, it reads the data and updates the consumer position variable in the block metadata. The two variants of *get* differ in which position variables and boundaries they use.  $get^f$  uses  $f\_pos$  as consumer position variable, NE as block margin, and  $b\_pos$  as boundary of valid data.  $get^b$  uses  $b\_pos$ , zero position of the block, and  $f\_pos$  for the same purposes, respectively.

Thieves follow a similar pattern: *steal* first checks if it has reached the block margin, or if the block has run out of data ( $s\_pos$  has reached  $b\_pos$ ). Then, it updates  $s\_pos$  using an atomic compare-and-swap (CAS) to point to the next entry, reads the data, and finally updates  $s\_cnt$  with an atomic increment. If the CAS fails, *steal* returns *conflict*. (CAS is used because multiple thieves can operate in the same block.)

All of these operations return *block\_done* when they reach a block margin. Otherwise, if the block runs out of data, *get* and *steal* return *empty*.

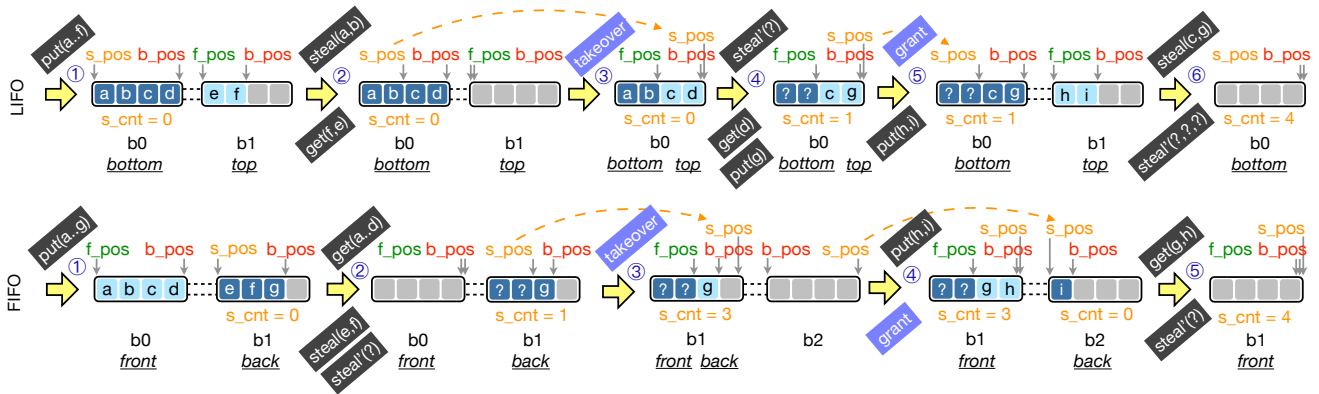


Figure 7: Takeover and grant procedures in block advancement.

## 4.2 Block Advancement

In case a block margin is reached, *put*, *get*, and *steal* move to the next block: They first check whether advancing is permitted by the round control, and if so, they call takeover (by *get*) or grant (by *put*) procedures, and reset block-level metadata.

### 4.2.1 Takeover and Grant Procedures

We explain the takeover and grant procedures using a queue with 4-entry blocks as an example (Fig. 7).

**LIFO.** Let us assume that 6 elements (a-f) were put into the queue. Thus, the owner is in the block  $b_1$ ;  $b\_pos$  in  $b_0$  and  $b_1$  becomes 4 and 2 respectively, while  $f\_pos$  and  $s\_pos$  remain at the initial value (0) (state ①). Then, two actions happen concurrently: two thieves try to steal entries, updating  $s\_pos$  in  $b_0$  to 2, and start to copy out the data (steal on Fig. 7), while the owner gets 3 values, consuming f, e (state ②), and advancing to  $b_0$ , thus starting the takeover. To perform the takeover, the owner atomically exchanges  $s\_pos$  with the block margin (4), and then sets  $f\_pos$  to the previous  $s\_pos$  value (2) (state ③). After the takeover, the owner gets d and puts g. Meanwhile, one ongoing *steal* completes (steal' on Fig. 7), increasing  $s\_cnt$  by 1 (state ④). It does not matter which of the two completes first. When the owner puts new items h and i, it grants  $b_0$  to thieves and advances to  $b_1$ . To perform the grant, it sets  $s\_pos$  to the  $f\_pos$  value (2), indicating to thieves that the block is available (state ⑤). After thieves steal all entries in  $b_0$ ,  $s\_cnt$  reaches the block margin (state ⑥). Thus,  $b_0$  can be reused in the next round.

**FIFO.** First, the producer puts 7 elements (a-g) into the queue. The producer and the consumer are in  $b_1$  and  $b_0$  respectively, and thieves can steal from  $b_1$  (state ①). Then, the consumer gets all elements in  $b_0$ , and advances to  $b_1$  (state ②). This requires taking over  $b_1$  from thieves: for this purpose, it updates  $s\_pos$  and  $f\_pos$  in the same way as the LIFO BWoS, but also adds the difference between the new  $f\_pos$  (2) and the block margin (i.e. length of the block) to  $s\_cnt$  (state ③). This way, when all thieves finish their operation in  $b_1$ , its  $s\_cnt$  will be equal to the block margin. After that, the producer puts a new item h, and advances to  $b_2$  granting it to thieves (state ④).

Finally, both thieves and the consumer have read all entries from  $b_1$ , its  $f\_pos$  and  $s\_cnt$  are equal to the block margin (state ⑤). The producer uses this condition to check if the block can be reused for producing new values into it.

### 4.2.2 Round Control and Reset Procedure

To implement round control (§3.3), the position variables in block metadata ( $f\_pos$ ,  $b\_pos$ ,  $s\_pos$ ,  $s\_cnt$ ) contain both the index or counter ( $idx$  field) as described in §4.2.1 and the round number ( $rnd$  field). We fit both components into a 64-bit variable that can be updated atomically.

Consider, for example, the put operation of FIFO BWoS (Fig. 8). In *put*, when the producer  $idx$  reaches the block margin NE of the block  $blk$  (step ①), the new round  $x$  of the next block  $nblk$  is calculated as described in §3.3 (step ②). When advancing to the block  $nblk$  with the producer round  $x$ , the producer checks that the consumer and the thieves have finished reading all data from the previous round in  $nblk$  by checking if their  $idx$  fields are equal to NE and their  $rnd$  fields are  $x - 1$  (step ③). When the check succeeds, the new value with the index 0 and the round  $x$  will be written into the producer position variable (step ④), thus *resetting* the block for the next round producing. Otherwise, a “queue full” condition is reported.

The *get* operation of the FIFO BWoS is similar. To decide whether *get* can use a next block, it checks whether the block’s next consumer’s round is equal to the producer round (step ③), and resets the round and index fields if the check succeeds.

Each operation resets only a subset of position variables ( $b\_pos$ ,  $f\_pos$ ,  $s\_pos$ ,  $s\_cnt$ ). We carefully select which variables each operation resets so that takeover and grant procedures by the owner have no write conflict with the reset done by thieves.

## 5 Verification and Optimization

The complexity of the BWoS algorithm necessitates the use of formal verification techniques to ensure that there are no lurking design or implementation bugs, and to optimize the

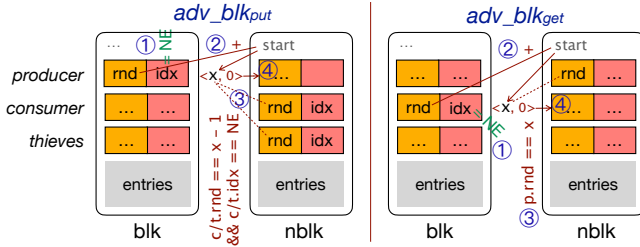


Figure 8: Round control in FIFO BWOs.

use on WMMs. One can easily imagine several tricky cases with block advancements. For example, for LIFO BWOs, when the owner calls *puts* and *gets* and advances to the next block, it may easily trigger ABA [67] bugs during the round control and takeover.

Unlike simpler algorithms like ABP [79], it is virtually impossible to justify the correctness of an optimal memory barrier placement by inspection. Luckily, model checking tools [62, 74, 84, 92] are widely used to check the correctness of concurrent algorithms and optimize the memory barrier under WMMs automatically, improving both performance and developer confidence. For example, the Tokio library uses the model checker Loom [27, 91], which has helped them find more than 10 bugs [28].

## 5.1 Verification Client

A model checker takes as input a small *verification client* program that invokes queue operations. It verifies that all possible executions of the input program satisfy some generic correctness properties, such as memory safety and termination [45], as well as any algorithm-specific properties that are included in the verification client as *assertions*. Whenever verification fails, the model checker returns a concrete erroneous execution as a counterexample.

To be able to generalize the verification result beyond the specific client program verified, the client program must trigger all possible contending scenarios and cover all desired properties. Because of the symmetry of BWOs (each owner operates on its own queue and steals from others), it suffices to verify the use of one queue owned by one thread and contended by several thief threads.

**Verified properties.** We have verified the following properties with the GenMC model checker [74, 75]:

- Memory safety: The program does not access uninitialized, unallocated or deallocated memory.
- Data race freedom: there are no data races on variables that are marked as non-atomic.
- Consistency: Each element written by the producer is read only once by either the consumer or thieves. No data corruption or loss occurs.
- Loop termination: Every unbounded spinloop and bounded fail-retry-loop in the program will eventually terminate even under weak memory models.

```

1 class stat {
2   u64 sum = 0, buf = 1;
3   void put(queue<u64> q){
4     if (q.put(buf))
5       sum += buf;
6     buf <<= 1;
7   }
8   bool get(queue<u64> q){
9     data = q.get(buf);
10    if (data != null) {
11      sum += data;
12      return true;
13    }
14    return false;
15  }
16  void steal(queue<u64> q){
17    data = q.steal(buf);
18    if (data != null)
19      sum += data;
20  }
21  }
22  stat f, b, s1, s2;
23  queue<u64> q; // 2 * 2
24  T0: b.put(q)*3; f.get(q)*2;
25    data = q.get(buf);
26    b.put(q)*4; f.get(q)*3;
27    T1: s1.steal(q);
28    T2: s2.steal(q)*2;
29    T3: while (f.get(q));
30    assert (b.sum == f.sum +
31            s1.sum + s2.sum);
32  (T0 || T1 || T2) ; T3

```

Figure 9: Verification and optimization client code.

	VERI/OPT time	memory barriers				#executions explored
		#SEQ	#ACQ	#REL	#RLX	
LIFO BWOs	62 min.	0	2	2	14	1.39 M
FIFO BWOs	53 min.	0	3	3	16	1.43 M
ABP	16 min.	4	3	1	7	2.05 M

Table 2: Statistics of the verification and optimization.

All possible executions, including those that occur due to weak memory reordering under the IMM [94] and RC11 [78] memory models, have been explored, and the aforementioned properties hold for each of them. With GenMC we were able to verify safety properties and termination of loops, but not the properties of individual operations.

**Contending scenarios.** As in any model checking verification, our models have a limited size within which the above properties hold. The client code for verifying and optimizing *put*, *get*, *steal* operations of BWOs is shown in Fig. 9. We configure the queue to have two blocks, each with the capacity of two entries (line 23). It is thus sufficient to put 5 entries to trigger the queue wraparound. We then launch 3 threads that run in parallel: The owner thread T0 has 3 rounds of *put* and *get* (lines 24-26) with different numbers of entries, trying to trigger block advancement for both producers and consumers in each round. Thief threads T1 and T2 steal one and two entries respectively, and thus together with T0, they trigger the queue empty condition, takeover, grant, and reset procedures, as well as conflicts between thieves.

**Assertion and properties.** After threads T0–T2 exit, thread T3 gets all remaining entries, and asserts that the sum of *put* elements is equal to the sum of elements read via *get* and *steal* (lines 30-31). Notice that the elements are generated as powers of two (line 6), therefore this assertion ensures that *each element written by the producer has been read only once*.

## 5.2 Results

We have optimized and verified the C code of LIFO and FIFO BWOs with the VSync framework and the GenMC model

checker. We have also verified the ABP queue using our verification client as a baseline. The statistics are shown in Table 2, broken down by memory barrier type: sequentially consistent (SEQ), acquire (ACQ), release (REL), and relaxed (RLX, i.e. plain memory accesses).

For BWoS, barrier optimization and verification finished in about an hour on a 6-core workstation [59], with over 1 million execution explorations. For ABP, the checking finishes in 16 minutes. More executions are explored for ABP since thieves and owner synchronize for every operation, which brings more interleaving cases.

**Verification confidence.** By adding one thread and discovering that no further barriers were required, we conclude that further increasing the thread count is unlikely to discover some missing barrier. Hence, we can avoid the state space explosion that happens with larger thread counts. On the other hand, discovering that an existing barrier had to be stronger would have forced us to review the algorithm in general.

**Experience.** Model checking proved itself to be invaluable during BWoS’s development. For example, an early version of LIFO BWoS had a bug where thieves would reset the `s_pos` variable when advancing to their following block (`blk`). In the case when the owner is advancing to its preceding block which also happens to be `blk`, it would update `s_pos` in the takeover procedure, which conflicts with the thieves’ reset procedure, resulting in data loss. This data loss was detected by GenMC with the verification client assertion (lines 30-31). We have fixed it by delegating the thieves’ `s_pos` reset procedure to the owner, thus removing this conflict.

**Optimization.** For BWoS, most concurrent accesses are converted to relaxed barriers, with the few remaining cases being release or acquire barriers. For the owner’s fast path that determines the performance, we have only one release barrier in the FIFO BWoS. In contrast, the highly optimized ABP [79] contains many barriers. In particular, owner operations contain 2 sequentially consistent, 1 acquire, and 1 release barriers, which significantly degrade its performance.

We note that these optimization results are optimal: relaxing any of these barriers produces a counterexample. To further increase our confidence in the verification result, we added another thief thread stealing one entry, and checked the optimized BWoS with GenMC. BWoS passes the check in 3 days with around 200 million execution explorations.

**Barrier analysis.** LIFO BWoS does not contain any barriers in the fast path because the owner and the thieves do not synchronize within the same block. An acquire-release pair is related to `s_pos` in the owner’s slow path and thieves’ fast path that ensures the correctness of the takeover procedure. Another acquire-release pair is related to `s_cnt` which ensures the owner doesn’t overwrite ongoing reading when it catches up with a thieves’ block (wraparound case). For FIFO BWoS, besides the above barriers, since producer and thieves need to synchronize within a block, an additional acquire-release pair

in their fast path is required.

## 6 Evaluation

**Experimental setup.** We perform all experiments on two x86 machines connected via 10Gbps Ethernet link, each with 88 hyperthreads (x86) [71], and one Arm machine with 96 cores (arm) [70]. The operating system is Ubuntu 20.04.4 LTS with Linux kernel version 5.7.0.

### 6.1 Block Size and Memory Overhead

In comparison with other queues, BWoS has extra parameters that the user needs to choose when initializing a data structure, namely the block size and the number of blocks. In our experience with both micro- and macro-benchmarks, the system’s throughput remains mostly constant regardless of the block size or the number of blocks as long as they are above certain minimal values: 8 or more blocks in the queue and 64 or more elements in the block, both for our x86 and arm machines.

The reason for this insensitivity to block size change is twofold: first, since a single thread is responsible for advancing the blocks of its own queue, the block size does not introduce any contention-related overhead. Larger block sizes cause the queue owner to advance the block less often, but after a certain block size, the overhead of advancing the block becomes negligible. Second, since BWoS forbids the owner and thieves consuming items in the same block with block-level synchronization, the contention of them on a queue is largely independent from the number of blocks. These insights guide the block size selection for our benchmarks: we set the number of blocks to 8 and calculated the block size based on the queue capacity.

Therefore, selecting an appropriate block size is straightforward. Further fine-tuning of these parameters may be beneficial for extreme scenarios where memory-size constraints are present or the overly large block size becomes detrimental to stealing (§8).

BWoS contains three pointers for each queue, and four atomic variables, two pointers, and one boolean variable for each block as its metadata. The actual memory usage also includes cache padding added to prevent false sharing. The memory overhead from this metadata is static and thus negligible for most use-cases.

### 6.2 Microbenchmarks

To verify our claims, we have designed a microbenchmark which supports both LIFO and FIFO work stealing and compared BWoS with the state-of-the-art algorithms: an off-the-shelf ABP [48] implementation from Taskflow v3.4.0 [35] with barrier optimization [79] (`abp`), the block-based bounded queue [106] (`bbq`), work stealing queues from Tokio v1.17.0 [38] (`tokioq`), Go’s runtime v1.18 [36]



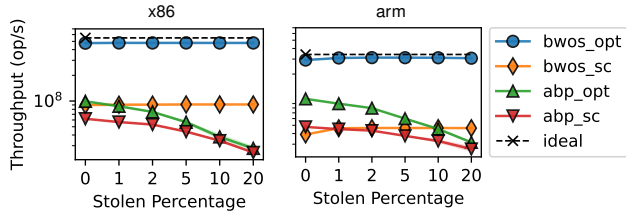


Figure 10: Throughput of LIFO BWoS and ABP with (opt) or without (sc) memory barrier optimization on x86 and arm running with different stolen percentages.

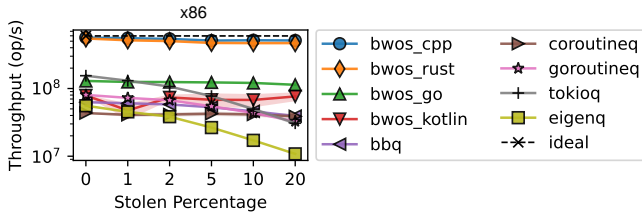


Figure 11: Throughput of FIFO BWoS and other state-of-the-art FIFO work stealing algorithms.

(goroutineq), Kotlin coroutines v1.6.4 [26] (coroutineq), and Eigen v3.3 [8] (eigenq).

Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks. We perform the following three experiments:

- Single queue without stealing (§6.2.1): The owner thread executes the workload in a loop: it first puts until the queue is full and then gets until the queue is empty.

- Single queue with stealing (§6.2.2): An additional thief thread calls *steal* operations on the queue in a loop. We adjust the *put/get* ratio, and the idle time between each *steal* to perform the experiment at varying stolen percentages<sup>2</sup>.

- Pool consisting of 8 queues (§6.2.3): 8 threads perform the following operations in a loop: put items to its queue until it is full, then get until it is empty, and then attempt to steal  $k * C$  items from the pool, where  $k$  is the *balancing factor* (in percent), and  $C$  is the queue capacity. The threads are distributed equally between two NUMA domains, and within each NUMA domain between two L3 cache groups [58].

In each experiment, we measure the total throughput: the sum of *put*, *get*, and *steal* operation throughputs (ops/sec).

### 6.2.1 Queue without Stealing

**Overall performance.** Figures 10 and 11 for stolen percentage equal to 0 show the performance of the queue without stealing. BWoS outperforms other algorithms by a significant margin. For example, LIFO BWoS (bwos\_opt) has 4.55x higher throughput than ABP (abp\_opt) on x86, and FIFO BWoS written in C/C++, Rust, Go, and Kotlin outperform bbq in C, eigenq in C++, tokioq in Rust, goroutineq in Go,

<sup>2</sup>The thief thread is located in the same L3 cache group as the owner; the results are similar when putting the thief thread elsewhere.

coroutineq in Kotlin by 8.9x, 10.15x, 3.55x, 1.61x, and 1.82x accordingly.

**Impact of the memory barrier optimization.** abp and LIFO BWoS get 1.65x and 5.39x speedup on x86, and 2.03x and 3.38x speedup on arm respectively due to the memory barrier optimization. We observe similar results for FIFO work stealing algorithms<sup>3</sup>. The much greater speedup of BWoS compared to ABP is possible in particular due to the separation of fast path and block advancement, where most of the barriers in the fast path become relaxed.

**Effectiveness of the block-level synchronization.** Results show that on x86 LIFO and FIFO BWoS are only 10.7% and 5.4% slower than ideal, respectively. On arm the results are similar. Thus, block-level synchronization allows BWoS to approach the theoretical upper bound by removing the consumer-thief synchronization from the fast path.

### 6.2.2 Queue with Stealing

**Overall performance.** As the stolen percentage increases, BWoS continues to outperform other work-stealing algorithms. For example, with 10% stolen percentage, LIFO BWoS outperforms abp by 12.59x, while FIFO BWoS outperforms bbq, eigenq, tokioq, goroutineq, coroutineq by 11.2x, 30.1x, 9.41x, 2.78x, and 1.64x respectively.

**Effectiveness of the block-based approach.** Unlike other algorithms, BWoS suffers only a minor performance drop as the stolen percentage increases. For example, for 20% stolen percentage, the throughput of LIFO and FIFO BWoS drops only by 0.53% and 9.35%, while for abp\_opt, tokioq and goroutineq it degrades by 71.9%, 80.2%, and 59.3% respectively. Note that the BBQ concurrent FIFO queue [106], which is also a block-based design, does not reach performance comparable to BWoS, stressing the importance of our design decisions for the work stealing workloads.

### 6.2.3 Pool with Different Stealing Policies

**Stealing policies.** We perform this experiment with 6 stealing policies, namely the random choice policy (rand), a policy that chooses the victim based on a static configuration (seq), a policy that chooses the last selected one as the victim [104] (last), best of two (best\_of\_two), best of many (best\_of\_many), and NUMA-aware policy (numa). For best\_of\_many we choose best of half (i.e. best of four).

**Overall performance.** In this experiment, we compare BWoS only with the second-best algorithm from the previous experiments: abp and tokioq for LIFO and FIFO work stealing respectively. Fig. 12 shows that BWoS performs consistently better than other algorithms. When the balancing factor is 0%, BWoS outperforms abp by 4.69x and tokioq by 2.68x. As the

<sup>3</sup>Notice here bwos\_go does not have barrier optimization because Go does not expose an interface for relaxed atomics. However, for the macrobenchmarks we apply the barrier optimization by using the Go internal atomic library.

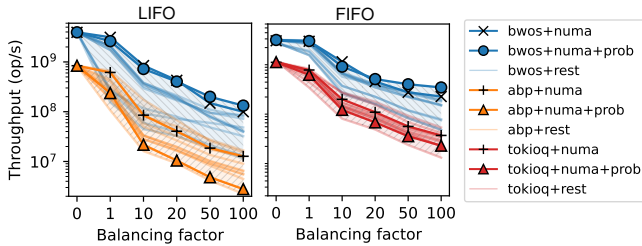


Figure 12: Throughput of the pool (8 queues) with different stealing policies and different balancing factors on x86. rest refers to all non-*numa* policies with and without probabilistic stealing.

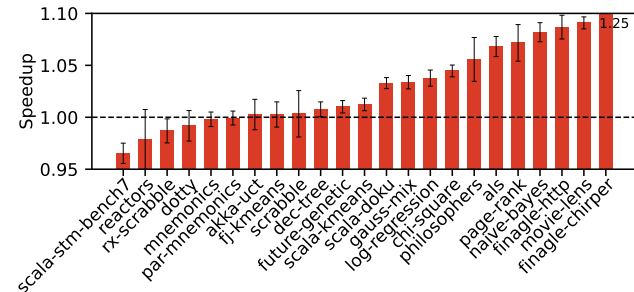


Figure 13: Speedup of 23 benchmarks from Renaissance benchmark suite on x86.

balancing factor increases, the throughput of BWoS variants is 7.90x higher than of abp and 6.45x higher than of tokioq.

**Impact of the NUMA-aware policy.** LIFO and FIFO BWoS with *numa* policy outperform BWoS with other policies by at most 2.21x and 1.73x respectively. For other work stealing algorithms, *best\_of\_two* brings the best performance. Thus, BWoS benefits from *numa* policy while other algorithms do not. On the other hand, in many cases *best\_of\_many* brings the worst performance, proving that interference with the owner can outweigh its improvements to the load balance.

**Effectiveness of the probabilistic stealing.** BWoS can additionally benefit from the probabilistic stealing. When the balancing factor is 100%, *numa* with probabilistic stealing (*bwos+numa+prob*) brings 1.34x, 1.53x performance improvement on average to LIFO and FIFO BWoS.

## 6.3 Macrobenchmarks

### 6.3.1 Java G1GC

We replace the task queue [24] in Java 19 HotSpot [37] with LIFO BWoS, and run the Renaissance benchmark suite v0.14.0 [33], which consists of 25 modern, real-world, and concurrent benchmarks [95] designed for testing and optimizing garbage collectors. Two database benchmarks are omitted since they don't support JDK 19. JVM enables `-XX:+DisableExplicitGC` [30,68] and `-XX:+UseG1GC` flags when running the benchmark. All other parameters (e.g., number of GC threads, VM memory limit) are default. We run 10 iterations for each benchmark with the modified and the

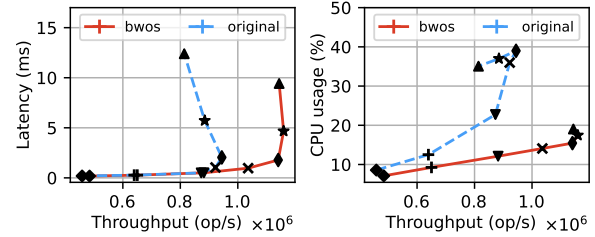


Figure 14: Throughput and latency results of Hyper HTTP server with BWoS and the original algorithm.

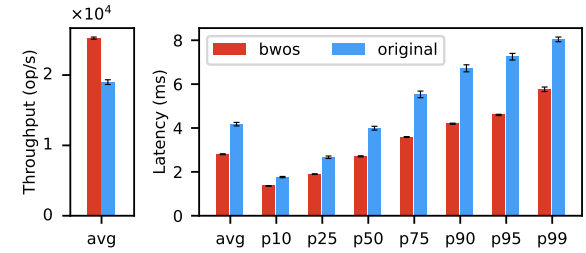


Figure 15: Throughput and latency of Tonic gRPC server with BWoS and the original algorithm.

original JVM, and measure the end-to-end program run time via the Renaissance testing framework.

Figure 13 shows the speedup of all 23 benchmarks on x86. When BWoS is enabled, 17 of them get performance improvement. The average speedup of all benchmarks is 3.55% and the maximum speedup is 25.3%. The applications that benefit more from concurrent GC also get greater speedup from BWoS. Results on arm are similar where the average speedup is 5.20%, 18 benchmarks are improved and the maximum speedup is 17.2%.

On the other hand, several Renaissance benchmarks did not get any performance improvement from using BWoS. We have investigated this issue by running JVM with flags `-Xlog:gc+cpu` and `-Xlog:gc+heap+exit` to collect GC-related statistics. These experiments have shown that applications that trigger GC often demonstrate improvement from BWoS, while applications that don't trigger GC or triggered it only rarely (e.g. at JVM exit) see no speedup. For the benchmarks which never or seldomly trigger the GC, the slowdown is most likely due to the longer queue initialization.

### 6.3.2 Rust Tokio Runtime

We replace the run queue [39] in Tokio v1.17.0 [38] with FIFO BWoS, and run Hyper HTTP server v0.14.18 [20] and Tonic gRPC server v0.6.2 [40] with the modified runtime. Tokio runtime (also Go runtime) provides a batch stealing interface. Based on observations from benchmarks similar to Fig. 2d, we configured the thief of BWoS to steal all available entries from its block at once. Benchmarks are performed on two x86 machines, one running the server, the other running the HTTP benchmarking tool *wrk* v4.2.0 [43] or the gRPC benchmarking and load testing tool *ghz* v0.017 [14]. All parameters of Hyper and Tonic are default. Each benchmark runs 100 seconds and

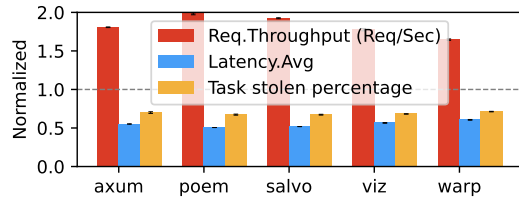


Figure 16: Request throughput, average latency, and task stolen percentage comparison results of 5 Rust web frameworks of BWoS (normalized to the original algorithm results) with rust-web-benchmarks workload on x86.

has 10 iterations. The latency and throughput are measured by wrk or ghz, while the CPU utilization of the server is collected through the Python psutil library [32]. wrk and ghz run the echo workload and SayHello protocol respectively and are configured to utilize all hyperthreads of their machine.

Figure 14 shows the throughput-latency and throughput-CPU utilization results of Hyper with different connection numbers (100, 200, 500, 1k, 2k, 5k, and 10k). Before the system is overloaded, BWoS provides  $1.14 \times 10^6$  op/s throughput while dropping 60.4% CPU usage with similar latency, the original algorithm provides only  $9.44 \times 10^5$  op/s throughput. With 1k connections, BWoS increases throughput by 12.3% with 6.74% lower latency and 60.9% lower CPU utilization.

Figure 15 shows the throughput and latency results of Tonic. Using BWoS increases throughput by 32.9%, with 32.8% lower average latency and 36.6% lower P95 latency.

To prove the generality of BWoS when applied to web frameworks, we also benchmark another 5 popular Rust web frameworks [4, 31, 34, 41, 42] that used Tokio runtime with rust-web-benchmarks [5] workload on x86 (Fig. 16). Results show that BWoS increases the throughput by 82.7% while dropping 45.1% of average latency. In addition, the task stolen percentage drops from 69.0% to 49.2%. We have made our implementation for the Tokio runtime available to the open-source community [3].

### 6.3.3 Go Runtime

We replace the runqueue [17] in the Go programming language [36] v1.18.0 runtime with BWoS and benchmark 9 JSON libraries [1, 9–12, 15, 18, 19, 25]. The benchmark suite [16] comes from the go-json library and runs 3 iterations with default parameters. We record the latency of each operation (e.g., encoding/decoding small/medium/large JSON objects) reported by the benchmark suite, and calculate the speedup.

As shown in Fig. 17, when BWoS is enabled, operations get 25.8% average performance improvement on x86. arm produces similar results with 28.2% speedup on average. In general, encoding operations have better speedup compared to decoding operations. We observe no improvement for encoding booleans and integers.

## 7 Related Work

**Block-based queues.** Wang *et al.* proposed a block-based bounded queue [106] (BBQ) that splits the buffer into multiple blocks, thus reducing the producer-consumer interference. BWoS differs from BBQ in the following ways: (1) although BBQ also applies metadata separation, the producer-consumer interference it reduces is not an issue for work stealing as these always execute on the same core. By introducing block-level synchronization, steal-from-middle property, and randomized stealing, FIFO BWoS outperforms BBQ by a large margin (§6). (2) For the round control in BWoS, the new round of a block is determined only by the round of its adjacent block instead of relying on global metadata, as the *version* mechanism in BBQ does. This design simplifies the round updating and reduces its overhead.

**Owner-thief interference and synchronization costs.** Attiya *et al.* proved that work stealing in general requires strong synchronization between the owner and thieves [49]. BWoS overcomes this issue by delegating this synchronization to the block advancement, thus removing it from the fast path. Acar *et al.* used a sequential deque with message passing to remove the owner’s barrier overhead [44]. However, this design relies on explicit owner-thief communication, thus the steal operation cannot run to completion in parallel with the owner’s operations. Dijk *et al.* proposed a deque-based LIFO work-stealing algorithm which splits the deque into owner and thief parts, thus reducing the owner’s memory fences when they do not reach the queue split point [61]. However, the entries read by thieves cannot be reused until the whole deque is empty. Horie *et al.* proposed a similar idea, where each owner has a public queue that is accessible from other threads and a private queue that is only accessible by itself [68]. However, it requires more effort to deal with load balancing, e.g., introducing global statistics metadata which causes more cache misses for the owner. In contrast, BWoS reduces the interference using techniques of block-level synchronization, and probabilistic and randomized stealing. Morrison *et al.* introduced work stealing algorithms which rely on the bounded TSO microarchitectural model, which x86 and SPARC CPUs were shown to possess [89]. Michael *et al.* reduced the thief-owner synchronization by allowing them to read the same task [87], which requires reengineering of tasks to be idempotent. BWoS exhibits correct and efficient execution on a wide range of CPU architectures without any additional requirements.

**Stealing policies.** Yang *et al.* gave a survey of scheduling parallel computations by work stealing [107]. Kumar *et al.* benchmarked and analyzed variations of stealing policies [76]. Mitzenmacher proposed to give the thief two choices for selecting the victim to have a better load balancing [88]. Most of the analyzed policies are size-based, and thus aim to reach the same goal as our probabilistic stealing policy—namely, better load balance. Hendler *et al.* allow thieves to steal half

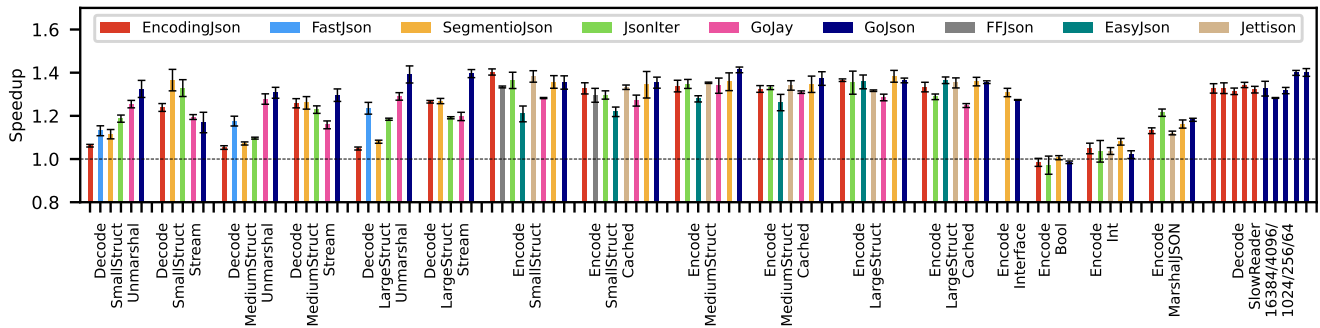


Figure 17: Speedup in the go-json library benchmark on x86 from using BWoS.

of the items in a given queue once to reduce interference [66]. BWoS supports batched stealing, but the maximum amount of data that can be stolen atomically is a block. However, the stealing policy can be configured to steal more than one block. Kumar *et al.* proposed a NUMA-aware policy for work stealing [77]. This policy is fully orthogonal to BWoS and can be combined with its probabilistic stealing policy.

**Formally verified work stealing.** Lê *et al.* [79] manually verified and optimized the memory barriers of Chase-Lev dequeue [53] on WMMs. Unlike the verification of BWoS which relies on model checking, manual verification is a high-effort undertaking. In the context of concurrent queues, Meta’s FollyQ was verified using interactive theorem prover [105]. While this approach provides the highest levels of confidence in the design, it works only with sequentially consistent memory model, and is also a high-effort endeavor. Recently, GenMC authors have verified the ABP queue as part of evaluation of their model checker [74]. The authors of BBQ have relied on VSync to simultaneously verify and optimize the barrier for weak memory models [106]. BWoS also uses VSync for this purpose, but instead of many hand-crafted tests, which exercise the individual corner cases in BBQ, we create one comprehensive client that covers several corner cases and their interactions at once. We further verify the optimization results by adding one more thief into the verification client and checking it with GenMC.

## 8 Conclusion

To conclude, we explore two of our learnings from this work.

**The benefit of the block-based design is manifold.** First, by replacing the global mutable metadata with block-level metadata, it is possible to eliminate the interference between the owner and the thieves that operate on different blocks. Second, by ensuring exclusive access to a block for owner’s get operation through block-level synchronization, it is possible to relax most of the barriers from the operation’s fast path, increasing its performance up to the theoretical upper bound. Although being unnecessary in our current algorithm, a third benefit is the verification modularity given by the block-based design, *e.g.*, allowing the verification of blocks

and their composition in separate steps. Finally, the block-based design opens possibilities for holistic optimization of the data structure use, as we do with our probabilistic stealing policy.

BWoS can also be applied to GPU and hybrid CPU-GPU computations, as well as in HPC schedulers, where work stealing is common. We plan to explore this direction in the future. More generally, the BWoS design can be applied to other use cases, where the data structure is mostly accessed by a single thread, and only rarely by multiple. In this case, the decisions demonstrated in BWoS can act as design and implementation guidelines.

### Verified software can be faster than unverified software.

The more hardware details and tweaks are mirrored in the software, the more complex and opaque that piece of code becomes. The interaction of this complexity with concurrency and weak memory consistency is a major challenge. We believe that practical verification tools (*i.e.*, tools applied to increase confidence in correctness) are a key enabler in the development of efficient, and inevitably complex, concurrent software such as BWoS.

### Future Work

There are several directions for further work: We plan to contribute BWoS to more open-source projects, *e.g.*, openJDK [23, 29], and Golang, as well as investigate how to use BWoS in HPC runtimes. We also plan to better explore the performance trade-offs for BWoS: if the number of outstanding work items is smaller than the block size, BWoS can prevent stealing and thus limit the achieved parallelism. Furthermore, if the queue capacity has to be very small (due to space requirements), it may be necessary to reduce the block size and thus incur more block advancement that leads to performance drop. These situations would benefit from more exploration in the system design. In other cases, BWoS is expected to outperform existing state-of-the-art work-stealing algorithms due to its implementation of several performance-enhancing techniques.

## Acknowledgments

We thank our shepherd Phillip Gibbons and the anonymous reviewers for their insightful comments.



## References

- [1] A high-performance 100% compatible drop-in replacement of "encoding/json". <https://github.com/json-iterator/go>.
- [2] ABA in local queue. <https://github.com/tokio-rs/tokio/issues/5041>.
- [3] Add BWoS-queue backend to tokio. <https://github.com/tokio-rs/tokio/pull/5283>.
- [4] axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. <https://github.com/tokio-rs/axum>.
- [5] Benchmarking web frameworks written in rust with rewrk tool. <https://github.com/programatik29/rust-web-benchmarks>.
- [6] C++ Atomic operations library. <https://en.cppreference.com/w/cpp/atomic/atomic>.
- [7] Cascade Lake - Microarchitectures - Intel. [https://en.wikichip.org/wiki/intel/microarchitectures/cascade\\_lake](https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake).
- [8] Eigen: a C++ template library for linear algebra. <https://eigen.tuxfamily.org/>.
- [9] Fast JSON encoder/decoder compatible with encoding/json for Go. <https://github.com/goccy/go-json>.
- [10] Fast JSON parser and validator for Go. <https://github.com/valyala/fastjson>.
- [11] Fast JSON serializer for golang. <https://github.com/mailru/easyjson>.
- [12] faster JSON serialization for Go. <https://github.com/pquerna/ffjson>.
- [13] Garbage First Garbage Collector Tuning. <https://www.oracle.com/technical-resources/articles/java/g1gc.html>.
- [14] ghz: gRPC benchmarking and load testing tool. <https://github.com/bojand/ghz>.
- [15] Go package containing implementations of efficient encoding, decoding, and validation APIs. <https://github.com/segmentio/encoding>.
- [16] GoJson benchmarks. <https://github.com/goccy/go-json/tree/master/benchmarks>.
- [17] golang run-queue. <https://github.com/golang/go/blob/master/src/runtime/proc.go>.
- [18] high performance JSON encoder/decoder with stream API for Golang. <https://github.com/francoispqt/gojay>.
- [19] Highly configurable, fast JSON encoder for Go. <https://github.com/wI2L/jettison>.
- [20] Hyper: An HTTP library for Rust. <https://github.com/hyperium/hyper>.
- [21] Intel Memory Latency Checker v3.9a. <https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html>.
- [22] Intel oneAPI Threading Building Blocks. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>.
- [23] Java Development Kit. <https://jdk.java.net/>.
- [24] JDK task queue. <https://github.com/openjdk/jdk/blob/master/src/hotspot/share/gc/shared/taskqueue.hpp>.
- [25] json package - encoding/json. <https://pkg.go.dev/encoding/json>.
- [26] Library support for Kotlin coroutines. <https://github.com/Kotlin/kotlinx.coroutines>.
- [27] Loom: Permutation testing for concurrent code. <https://docs.rs/crate/loom/0.2.4>.
- [28] Making the Tokio scheduler 10x faster. <https://tokio.rs/blog/2019-10-scheduler>.
- [29] OpenJDK. <https://openjdk.org/>.
- [30] Performance Tuning Guide. <https://docs.oracle.com/cd/E19159-01/819-3681/abeih/index.html>.
- [31] Poem Framework: A full-featured and easy-to-use web framework with the Rust programming language. <https://github.com/poem-web/poem>.
- [32] psutil - PyPI. <https://pypi.org/project/>.
- [33] Renaissance Suite. <https://renaissance.dev/>.
- [34] Salvo: A powerful and simplest web server framework in Rust world. <https://github.com/salvo-rs/salvo>.
- [35] Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. <https://github.com/taskflow/taskflow>.
- [36] The Go programming language. <https://go.dev/>.

- [37] The HotSpot Group. <http://openjdk.java.net/groups/hotspot>.
- [38] Tokio: A runtime for writing reliable asynchronous applications with Rust. <https://github.com/tokio-rs/tokio>.
- [39] Tokio run-queue. [https://github.com/tokio-rs/tokio/blob/master/tokio/src/runtime/scheduler/multi\\_thread/queue.rs](https://github.com/tokio-rs/tokio/blob/master/tokio/src/runtime/scheduler/multi_thread/queue.rs).
- [40] Tonic: A native gRPC client & server implementation with async/await support. <https://github.com/hyperium/tonic>.
- [41] Viz: Fast, flexible, lightweight web framework for Rust. <https://github.com/viz-rs/viz>.
- [42] warp: A super-easy, composable, web server framework for warp speeds. <https://github.com/seanmonstar/warp>.
- [43] wrk: Modern HTTP benchmarking tool - GitHub. <https://github.com/wg/wrk>.
- [44] ACAR, U. A., CHARGUÉRAUD, A., AND RAINEY, M. Scheduling parallel programs by work stealing with private dequeues. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2013), pp. 219–228.
- [45] ALPERN, B., AND SCHNEIDER, F. B. Defining liveness. *Information processing letters* 21, 4 (1985), 181–185.
- [46] AMAZON WEB SERVICES. AWS Graviton Processor – Enabling the best price performance in Amazon EC2, 2020. <https://mysqleonarm.github.io/ARM-LSE-and-MySQL/>.
- [47] ARNAUTOV, S., FELBER, P., FETZER, C., AND TRACH, B. FFQ: A fast single-producer/multiple-consumer concurrent FIFO queue. In *2017 IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2017)* (2017), pp. 907–916.
- [48] ARORA, N. S., BLUMOFÉ, R. D., AND PLAXTON, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems* 34, 2 (2001), 115–144.
- [49] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. T. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 487–498.
- [50] BLUMOFÉ, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.
- [51] BLUMOFÉ, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.
- [52] CEDERMAN, D., AND TSIGAS, P. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), pp. 57–64.
- [53] CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (2005), pp. 21–28.
- [54] CHATTERJEE, S., GROSSMAN, M., SBÎRLEA, A., AND SARKAR, V. Dynamic task parallelism with a GPU work-stealing runtime system. In *International Workshop on Languages and Compilers for Parallel Computing* (2011), Springer, pp. 203–217.
- [55] CHO, I., SAEED, A., FRIED, J., PARK, S. J., ALIZADEH, M., AND BELAY, A. Overload Control for  $\mu$ -scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 299–314.
- [56] CONG, G., KODALI, S., KRISHNAMOORTHY, S., LEA, D., SARASWAT, V., AND WEN, T. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing* (2008), IEEE, pp. 536–545.
- [57] CONWAY, M. E. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, fall joint computer conference* (1963), pp. 139–146.
- [58] DE LIMA CHEHAB, R. L., PAOLILLO, A., BEHRENS, D., FU, M., HÄRTIG, H., AND CHEN, H. CLOF: A compositional lock framework for multi-level NUMA systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 851–865.
- [59] DELL. Precision 5820 Tower Spec. [https://i.dell.com/sites/csdocuments/Shared-Content\\_data-Sheets\\_Documents/en/us/Precision-5820-Tower-Spec-Sheet.pdf](https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/us/Precision-5820-Tower-Spec-Sheet.pdf).
- [60] DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management* (2004), pp. 37–48.

- [61] DIJK, T. V., AND POL, J. C. Lace: non-blocking split deque for work-stealing. In *European Conference on Parallel Processing* (2014), Springer, pp. 206–217.
- [62] GAVRILENKO, N., PONCE-DE LEÓN, H., FURBACH, F., HELJANKO, K., AND MEYER, R. BMC for weak memory models: Relation analysis for compact SMT encodings. In *International Conference on Computer Aided Verification* (2019), Springer, pp. 355–365.
- [63] HALSTEAD JR, R. H. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 9–17.
- [64] HARRIS, T., AND KAESTLE, S. Callisto-RTS: Fine-grain parallel loops. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 45–56.
- [65] HARRIS, T., MAAS, M., AND MARATHE, V. J. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), pp. 1–14.
- [66] HENDLER, D., AND SHAVIT, N. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), pp. 280–289.
- [67] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, USA, 2011.
- [68] HORIE, M., HORII, H., OGATA, K., AND ONODERA, T. Balanced double queues for GC work-stealing on weak memory models. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (2018), pp. 109–119.
- [69] HORIE, M., OGATA, K., TAKEUCHI, M., AND HORII, H. Scaling up parallel GC work-stealing in many-core environments. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (2019), pp. 27–40.
- [70] HUAWEI. 2280 Balanced Model - Huawei Enterprise. <https://e.huawei.com/uk/products/servers/taishan-server/taishan-2280-v2>.
- [71] HUAWEI. FusionServer Pro 2288X V5 Rack Server. <https://support-it.huawei.com/server-3d/res/server/2288xv5/index.html?lang=en>.
- [72] KNUTH, D. E. *The Art of Computer Programming*, vol. 3. Pearson Education, 1997.
- [73] KOGIAS, M., PREKAS, G., GHOSN, A., FIETZ, J., AND BUGNION, E. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019), pp. 863–880.
- [74] KOKOLOGIANNAKIS, M., RAAD, A., AND VAFEIADIS, V. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, pp. 96–110.
- [75] KOKOLOGIANNAKIS, M., AND VAFEIADIS, V. GENMC: A model checker for weak memory models. In *International Conference on Computer Aided Verification* (2021), Springer, pp. 427–440.
- [76] KUMAR, S., AND SAHU, A. Benchmarking and analysis of variations of work stealing scheduler on clustered system. In *2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies* (2014), IEEE, pp. 28–35.
- [77] KUMAR, V. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)* (2020), IEEE, pp. 251–260.
- [78] LAHAV, O., VAFEIADIS, V., KANG, J., HUR, C., AND DREYER, D. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), A. Cohen and M. T. Vechev, Eds., ACM, pp. 618–632.
- [79] LÊ, N. M., POP, A., COHEN, A., AND NARDELLI, F. Z. Correct and efficient work-stealing for weak memory models. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2013* (2013), Association for Computing Machinery, pp. 69–79.
- [80] LEA, D. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), pp. 36–43.
- [81] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. *Acm Sigplan Notices* 44, 10 (2009), 227–242.
- [82] LI, J., DINH, S., KIESELBACH, K., AGRAWAL, K., GILL, C., AND LU, C. Randomized work stealing for large scale soft real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)* (2016), IEEE, pp. 203–214.

- [83] LIU, C., SONG, P., LIU, Y., AND HAO, Q. Efficient work-stealing with blocking deque. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICSS)* (2014), IEEE, pp. 149–152.
- [84] LORCH, J. R., CHEN, Y., KAPRITSOS, M., PARNO, B., QADEER, S., SHARMA, U., WILCOX, J. R., AND ZHAO, X. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020), pp. 197–210.
- [85] MARTÍNEZ, M. A., FRAGUELA, B. B., AND CABALEIRO, J. C. A parallel skeleton for divide-and-conquer unbalanced and deep problems. *International Journal of Parallel Programming* 49, 6 (2021), 820–845.
- [86] MCCLURE, S., OUSTERHOUT, A., SHENKER, S., AND RATNASAMY, S. Efficient scheduling policies for microsecond-scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 1–18.
- [87] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2009), pp. 45–54.
- [88] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.
- [89] MORRISON, A., AND AFEK, Y. Fence-free work stealing on bounded TSO processors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014* (2014), R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, pp. 413–426.
- [90] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013* (2013), A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds., ACM, pp. 131–150.
- [91] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications* (2013), pp. 131–150.
- [92] OBERHAUSER, J., CHEHAB, R. L. D. L., BEHRENS, D., FU, M., PAOLILLO, A., OBERHAUSER, L., BHAT, K., WEN, Y., CHEN, H., KIM, J., AND VAFEIADIS, V. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, pp. 530–545.
- [93] OUYANG, K., SI, M., HORI, A., CHEN, Z., AND BALAJI, P. CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–15.
- [94] PODKOPEV, A., LAHAV, O., AND VAFEIADIS, V. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31.
- [95] PROKOPEC, A., ROSÀ, A., LEOPOLDSEDER, D., DUBOSCQ, G., TUMA, P., STUDENER, M., BULEJ, L., ZHENG, Y., VILLAZÓN, A., SIMON, D., ET AL. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 31–47.
- [96] QIAN, J., SRISA-AN, W., LI, D., JIANG, H., SETH, S., AND YANG, Y. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform*. 2015, pp. 170–181.
- [97] SCHMAUS, F., PFEIFFER, N., HÖNIG, T., NOLTE, J., AND SCHRÖDER-PREIKSCHAT, W. Nowa: A wait-free continuation-stealing concurrency platform. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), IEEE, pp. 360–371.
- [98] SCHWEIZER, H., BESTA, M., AND HOEFLER, T. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), IEEE, pp. 445–456.
- [99] STEINBERGER, M., KAINZ, B., KERBL, B., HAUSWIESNER, S., KENZEL, M., AND SCHMALSTIEG, D. Softshell: dynamic scheduling on GPUs.



*ACM Transactions on Graphics (TOG)* 31, 6 (2012), 1–11.

- [100] SUKSOMPONG, W., LEISERSON, C. E., AND SCHARDL, T. B. On the efficiency of localized work stealing. *Information Processing Letters* 116, 2 (2016), 100–106.
- [101] SUO, K., RAO, J., JIANG, H., AND SRISA-AN, W. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.
- [102] TOSS, J. Work stealing inside GPUs.
- [103] TZENG, S., PATNEY, A., AND OWENS, J. D. Task management for irregular-parallel workloads on the GPU.
- [104] VENNERS, B. The Java Virtual Machine. *Java and the Java virtual machine: definition, verification, validation* (1998).
- [105] VINDUM, S. F., FRUMIN, D., AND BIRKEDAL, L. Mechanized verification of a fine-grained concurrent queue from meta’s folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2022), pp. 100–115.
- [106] WANG, J., BEHRENS, D., FU, M., OBERHAUSER, L., OBERHAUSER, J., LEI, J., CHEN, G., HÄRTIG, H., AND CHEN, H. *BBQ*: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 249–262.
- [107] YANG, J., AND HE, Q. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming* 46, 2 (2018), 173–197.
- [108] ZHANG, Y., KUMAR, G., DUKKIPATI, N., WU, X., JHA, P., CHOWDHURY, M., AND VAHDAT, A. Ae-quitas: admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 1–18.



# Spoq: Scaling Machine-Checkable Systems Verification in Coq

*Xupeng Li*  
Columbia University

*Xuheng Li*  
Columbia University

*Wei Qiang*  
Columbia University

*Ronghui Gu*  
Columbia University

*Jason Nieh*  
Columbia University

## Abstract

System software is often large and complex, resulting in many vulnerabilities that can potentially be exploited to compromise the security of a system. Formal verification offers a potential solution to creating bug-free software, but a key impediment to its adoption remains proof cost. We present Spoq, a highly automated verification framework to construct machine-checkable proofs in Coq for system software with much less proof cost. Spoq introduces a novel program structure reconstruction technique to leverage LLVM to translate C code into Coq, supporting full C semantics, including C macros, inline assembly, and compiler directives, so that source code no longer has to be manually modified to be verified. Spoq leverages a layering proof strategy and introduces novel Coq tactics and transformation rules to automatically generate layer specifications and refinement proofs to simplify verification of concurrent system software. Spoq also supports easy integration of manually written layer specifications and refinement proofs. We use Spoq to verify a multiprocessor KVM hypervisor implementation. Verification using Spoq required 70% less proof effort than the manually written specifications and proofs to verify an older implementation. Furthermore, the proofs using Spoq hold for the unmodified implementation that is directly compiled and executed.

## 1 Introduction

System software such as operating systems and hypervisors [7] forms the software foundations of our computing infrastructure. However, modern system software is large, complex, and imperfect, with vulnerabilities that can be exploited to compromise the security of a system. Formal verification offers a potential solution to this problem by mathematically proving that system software can provide critical security guarantees. This typically involves verifying that the software implementation satisfies a formal high-level specification of its behavior, then proving that the specification guarantees the desired security properties.

The former, referred to as functional correctness, is generally the most challenging part to do, given the complexity of system software implementations. Implementations are commonly written in C, which has complex semantics and language features, many unsupported by verification tools. Verification tools powerful enough to verify real-world system software are difficult and tedious to use to write specifications and proofs. Furthermore, a high-level specification that is useful for verifying higher-level properties such as security often has a significant semantic gap from the implementation, requiring substantial manual proof effort to bridge this gap. However, without functional correctness to ensure that the proofs hold on the actual implementation, formally verified guarantees can be meaningless in practice.

We introduce Spoq (Scaling Proofs in Coq), a new verification framework to reduce proof costs for machine-checkable verification of system software. Spoq focuses on simplifying formal verification of functional correctness to reduce proof costs while ensuring that all proofs are machine-checkable by a theorem prover and verified down to the actual software implementation. It operates on widely used unmodified C code and leverages the Coq proof assistant [55] to enable machine-checkable verification of complex systems. Its key feature is making Coq easier to use by automating many aspects of writing Coq specifications and proofs. This reduces the amount of Coq code that needs to be manually written, which significantly reduces the time to conduct machine-checkable verification.

Spoq is the first system that can automatically translate unmodified C systems code, such as found in the Linux kernel, into a Coq representation so that it can be verified. Previous approaches such as CompCert's ClightGen [35] only support a subset of the C language. Systems that use ClightGen such as CertiKOS [18, 20] require significant manual effort to retrofit the systems implementation before it can be verified, extra effort to develop and maintain the retrofitted version, and still cannot provide any verified guarantees on the actual running version. Spoq address this problem by leveraging the widely used Clang compiler front end to parse C code into

LLVM's language-independent intermediate representation (IR). Because LLVM IR represents functions as control flow graphs, Spoq introduces a novel program reconstruction technique that translates control flow graphs back into a Coq representation using program-style functions with if-then-else and loop statements that is more amenable to verification. This approach enables Spoq to support full C language semantics, including GNU C-specific extensions and inline assembly code, yet work with an IR with clean semantics designed for automated translation into another representation.

Spoq then leverages a layering proof strategy based on Concurrent Certified Abstraction Layers (CCAL) [19, 21] to modularize and decompose verification into smaller steps to make each verification step easier. This involves defining the layer structure of the implementation, where each layer consists of a group of functions that define the layer's interface. Higher layers can call the functions exposed by a lower layer's interface, but not the other way around. The top layer is a high-level specification of the behavior of the entire implementation, while the bottom layer is a machine model whose interface is designed to support LLVM IR semantics. Verification involves proving that the layers compositionally refine the top layer specification of the entire implementation. While layering makes each verification step easier to accomplish, if done manually, it has the disadvantage of requiring a user to construct additional layer specifications, including both low-level and high-level specifications, and refinement proofs for each layer, which can involve tediously writing thousands of lines of additional Coq code. That code then has to be manually rewritten each time the program implementation is updated, imposing significant, time-consuming proof costs. Spoq instead takes advantage of layering and the easier verification steps it affords to make it possible to automatically generate the Coq layer specifications and mechanized refinement proofs from the layer structure definition. It is the first system that can automate the generation of layered specifications and proofs in Coq for concurrent system software.

Spoq constructs a machine-checkable proof object for each layer showing its implementation built on top of a lower layer interface refines its own layer interface. It decomposes the proof for a layer into two tasks. The first task is to prove that the layer's implementation, namely its Coq abstract syntax tree (AST) representation, refines a low-level specification that is closer to the source code and independent of the state of the machine model. The second task is to prove that the low-level specification, built on top of a lower layer interface, refines a high-level specification that defines the layer's interface and is self-contained. By self-contained, we mean that the specification does not contain any calls to functions in any other layer other than the bottom layer machine model. Making the high-level specification self-contained simplifies verification because refinement proofs of any layers built on top of this layer can effectively ignore any layers below it.

Spoq introduces a library of Coq tactics to automatically

generate low-level specifications and refinement proofs between the implementation and low-level specification. Functions with loops are synthesized into Coq recursive specifications, then refined to their specifications using an induction proof template. To generate the specification for a function with loops, a ranking function is provided for each loop, which is monotonically decreasing and non-negative during loop iterations. Spoq leverages the ranking functions to generate loop termination proofs.

Spoq introduces transformation rules to automatically generate high-level specifications and refinement proofs between low-level and high-level specifications. Transformation rules include unfolding function definitions, syntactically reorganizing program structures, eliminating pre-determined branches and assertions, and performing mathematical simplification. Refinement proofs are done by introducing automatically generated annotations to track how transformations are applied, then using Coq tactics to prove the sequence of transformations preserves specification semantics. Automatic generation of specifications and proofs is only done for high-level specifications that do not introduce data abstractions to hide low-level data representation details, such as abstracting an array into a Coq `Map`. High-level specifications that introduce data abstractions or have very complex functions require manual assistance from the user to complete the specifications and proofs. Our experience indicates that the vast majority of functions can be automatically specified and refined without manual effort.

Spoq reduces the trusted computing base (TCB) for performing source code-level mechanized verification. There is no need to trust Spoq for generating specifications or proofs. Incorrect specifications will be rejected during refinement proofs, and incorrect proofs will be rejected by the Coq proof checker. Although Spoq relies on Clang which is not verified, most system software already needs to trust either widely used Clang or unverified alternatives such as the GNU C compiler to generate the executable code that actually runs. Using a verified compiler such as CompCert [35] is not viable in practice since it cannot even compile C code such as Linux kernel code. The only part of Spoq that is unverified yet needs to be trusted is its translator from LLVM IR to Coq, which is minimal by design. This TCB is much smaller than CompCert's ClightGen, which is larger and more complex since it has to directly parse and translate C code, a more difficult and involved process.

We have implemented Spoq and evaluated its effectiveness on commodity system software. We show that Spoq automatically translates over 99% of functions in unmodified C systems code into Coq representations, including the source code for the Linux kernel, while ClightGen fails to translate the vast majority of functions, including almost complete failure on the Linux kernel. We use Spoq to verify a multiprocessor KVM hypervisor implementation. Although an older version of the hypervisor was previously verified in Coq without Spoq, the proofs no longer work with the

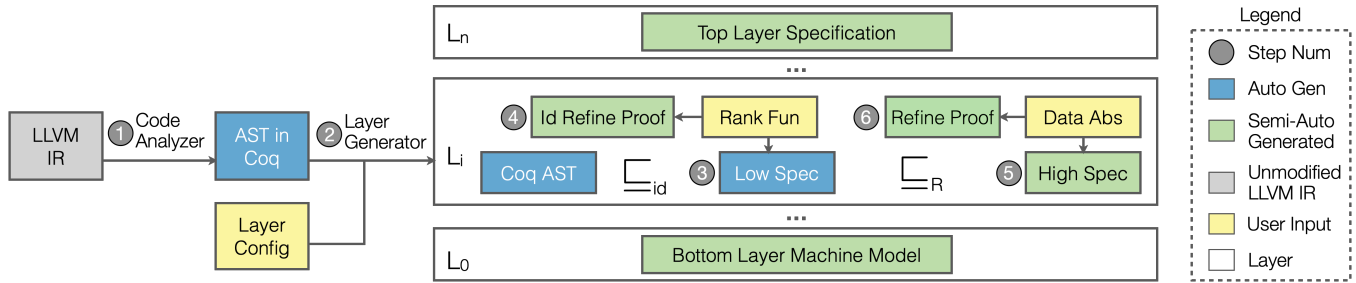


Figure 1: Spoq workflow.

updated version that supports additional hardware platforms. Previously, using ClightGen to translate the C implementation into a Coq representation required modifications to the source code, creating a gap between the verified and running code. Verifying the updated hypervisor using Spoq required much less proof effort, reducing the amount of manually written Coq code by over 70% compared to the verification of the older implementation. The proofs using Spoq are done on the unmodified source code of the hypervisor that is directly compiled and executed. Spoq even automatically generates the top layer specification, which we then use to verify the overall security properties of the hypervisor hold on the actual running software implementation.

## 2 Spoq Usage Model

To use Spoq, a user compiles the source code into LLVM IR and writes a layer configuration file defining the layer structure for the proof. The layer structure is defined to modularize the proof, with the additional constraint that a layer can only call functions in lower layers. For example, if the source code has three functions A, B and C such that A calls B and B calls C, at least three layers must be used. The configuration file specifies the name of each layer, the name of each function in each layer, the path to the source IR code, and the path to the Coq project. The configuration file should include the bottom layer abstract machine model, including its machine state definition. Spoq then generates the Coq project, including all specifications and proofs for each layer. If the source code or layer structure are changed, the user can rerun Spoq to update the Coq project. Spoq will regenerate the specifications and proofs for the parts affected by the changes, while other parts will remain unchanged.

Spoq guarantees that all generated specifications have exactly the same behavior as their source code implementations, but some generated high-level specifications may be too complex to be useful, and some refinement proofs may fail. Spoq makes it easy to integrate manually written specifications and proofs, which are simply annotated in the layer configuration file so that Spoq uses the provided specifications or proofs instead of generating them directly. If Spoq generates a high-level specification for a layer that is not concise enough, especially in how it updates the machine

state, the user can manually write the specification and rerun Spoq with the provided high-level specification. If Spoq fails in generating refinement proofs for a layer, the user will see the resulting compilation errors of the generated Coq project identifying the specific functions with errors. If the error occurs for a generated specification, it is most likely due to a failed loop termination proof. The user can manually write the loop termination proof that failed and rerun Spoq with the provided termination proof. If the error occurs for a manually written specification, the user can check if there is an error in the specification or if the refinement proof also needs to be manually written, then rerun Spoq again.

Spoq is useful for both verifying functional correctness as well as higher-level system properties such as security. In verifying functional correctness, Spoq can generate the top-level specification, which will be guaranteed to have exactly the same behavior as the source code implementation. This notion of functional correctness ensures that the implementation satisfies the specification, but not necessarily that the code has no bugs. If the code is buggy, the generated top-level specification will still have the same behavior, including any buggy behavior. To provide a stronger notion of correctness, a user can use the generated top-level specification to verify higher-level properties such as security, which will identify bugs in the specification. Alternatively, a user can manually write the top-level specification and leverage Spoq to generate intermediate layer specifications and refinement proofs to verify that the implementation is functionally correct with respect to a manually written specification, though such a specification can also have bugs. The key benefit of Spoq is ensuring that whatever verification is done holds not just for a specification, but all the way down to the source code implementation.

## 3 Spoq Workflow

Figure 1 shows the workflow of Spoq. We use the example in Figure 2 to explain each step in the workflow and show how Spoq scales machine-checkable verification for systems code. This example contains a simplified C function `alloc` to allocate a free page by scanning the array of page descriptors `page`. The main computation is implemented as a statement expression in a macro definition `ALLOC`, in which we use a loop to iterate all elements of `page` and set the page status of



```

// Layer interface L1
uint page[MAX_PAGE];
uint get_page (uint i) { return page[i] }
void set_page (uint i, uint s) { page[i] = s; }
// Layer interface L2
#define ALLOC() ( \
    uint i; \
    for (i = 0; i < MAX_PAGE; i++){ \
        if (get_page(i) == 0) { \
            set_page(i, 1); \
            break; \
        } \
    } \
    i;})
uint alloc() { return ALLOC(); }

```

**Figure 2:** A running example to allocate a free page.

the first free page to 1. The accesses to `page` are encapsulated into functions `get_page` and `set_page`. This coding style is quite common in systems software such as Linux kernel code.

**Generating Coq representations.** To conduct mechanized verification, the first step is to translate the implementation into a representation in theorem provers, which is challenging even for simple and common C systems code like `alloc`; ClightGen cannot parse this simple example. Spoq leverages the Clang compiler front end to parse C into LLVM IR, and provides a code analyzer to parse LLVM IR code into an AST representation defined in Coq (Step 1 in Figure 1). We use LLVM IR because it is language- and machine-independent, supports full C language semantics and most extensions of C, can be easily integrated with assembly code semantics, and is much simpler and more rigorously defined than C. However, LLVM IR does not keep program structures, such as if-then-else and loop statements, making it hard to conduct proofs in a structural and inductive manner. Spoq resolves this issue by analyzing the control flow graphs of the LLVM IR code and reconstructing program structures. For example, Spoq reconstructs the loop, branch, and break statements in the Coq representation for the LLVM IR generated from the `alloc` function in Figure 2:

```

Definition f_alloc :=
  { | fname := "alloc"; rettype := ...; fargs := ...;
    fbody := ... ::
      (ILoop (... :: (IIif ... IBreak) :: ...))... |}.

```

Spoq also models the semantics of Armv8 instructions [4] and parses assembly code into a list of assembly instructions in their Coq representations.

**Defining layer structure.** Spoq takes as input a layer configuration file which it uses to scale constructing mechanized proofs using CCALs. Using CCALs, we can construct a machine-checkable proof object “ $M@L \sqsubseteq_R L'$ ,” showing that the implementation  $M$ , built on top of a lower layer interface  $L$ , refines the interface  $L'$  with the refinement relation  $R$ . The file defines the layers and at which layer each function should be verified (Step 2 in Figure 1). For example, the layer configuration for the running example in Figure 2 defines that `get/set_page` should be verified on top of layer  $L_0$ , while `alloc` should be verified on top of layer  $L_1$ .

The layer structure presumes a bottom layer machine model, which Spoq automatically generates in part by identifying each global memory object in the source code and generating a corresponding machine state in Coq. Spoq also generates memory load/store primitives for each element in the state. The primitives take a memory pointer as an argument and calculate based on offset the array indices and structure elements to be accessed. Index boundary and data range checks are also included. The initial generated machine model does not include concurrency-related structures, such as an event log and oracle [40], which need to be manually added to complete the model to support CPU-local concurrency reasoning.

Given the layer configuration file, Spoq will automate generating the CCALs. It will build a CCAL “ $M_{\text{page}}@L_0 \sqsubseteq_{R_1} L_1$ ” to abstract the `page` array into a Coq Map object from natural numbers to integers, such that its elements can only be accessed through getter and setter methods, `get_page` and `set_page`, respectively, rather than arbitrary memory operations which may lead to unexpected behavior. The refinement relation  $R_1$  defines how the `page` array is abstracted into the Map object. It will then build a CCAL “ $M_{\text{alloc}}@L_1 \sqsubseteq_{id} L_2$ ” to verify the `alloc` function on top of  $L_1$  using the Map object without the need to worry about concrete implementation details of `page`. Here,  $id$  is an identical refinement relation since no data abstraction is needed when verifying `alloc`.

To make building CCALs easier, Spoq decomposes the required proofs into an *identical refinement* and a *lifting refinement*. The identical refinement refines  $M$  to a low-level specification  $S_{low}$  that is closer to the code and does not introduce any data abstraction, i.e., “ $M@L \sqsubseteq_{id} S_{low}$ .” The lifting refinement refines the low-level specification to a high-level specification  $L'$ , i.e., “ $S_{low} \sqsubseteq_R L'$ .” The high-level specification is self-contained and may introduce abstractions to some data in lower layers.

**Synthesizing identical refinements.** Spoq generates low-level specifications and identical refinement proofs for each layer. The low-level specification of a function aggregates the small-step transition of each instruction in the function into a big-step transition of the entire function while preserving the semantics. For assembly code and C code without loops, generating the specifications and proofs is straightforward (Step 3 in Figure 1). Spoq provides a Coq tactic library to generate the identical refinement proofs; a tactic is a pre-defined decision procedure to generate proof scripts in Coq. Neither the specification generator nor tactic library needs to be trusted, since incorrect low-level specifications will be rejected by refinement proofs, and incorrect proofs will be rejected by the Coq proof checker.

For C code with loops, Spoq requires the user to provide a ranking function for each loop, which is non-negative and monotonically decreasing during the loop iterations. This is necessary because a termination proof is needed for each loop to prove refinement, and automating such termination proofs without user input is generally undecidable. With the input

```

Definition rank (i: nat) := MAX_PAGE - i. (*user input*)
Fixpoint alloc_loop_low (r i: nat) (st: ST) :=
  match r with
  | 0 => Some (MAX_PAGE, st)
  | S r' =>
    match get_page_high i st with (* spec from L1 *)
    | Some 0 => match set_page_high i 1 st with
      | Some st' => Some (i, st')
      | None => None
    end
    | Some _ => alloc_loop_low r' (i+1) st
    | _ => None
  end
end.
Definition alloc_low (st: ST) :=
  let r := rank 0 in alloc_loop_low r 0 st.

```

Figure 3: Low-level specification for the alloc function.

ranking function, Spoq automatically synthesizes a recursive function as the low-level specification using the Fixpoint construction in Coq, which requires an argument that decreases for each recursive call. For example, Figure 3 shows a recursive function `alloc_loop_low` synthesized for the loop in the `alloc` function with a user-provided ranking function (`MAX_PAGE-i`) as the decreasing argument for the Fixpoint construction. Note that the low-level specification of `alloc` is not self-contained and depends on functions `get_page_high` and `set_page_high` provided by the high-level specification at a lower layer. Spoq generates the refinement proof using a uniform induction-proof template (Step 4 in Figure 1).

**Synthesizing lifting refinements.** Spoq generates high-level specifications and lifting refinement proofs for each layer. This is done automatically when data abstractions are not used to hide low-level data representation details to simplify proofs at higher layers. If data abstractions are needed, users need to formulate the refinement relations, define abstract operations, and conduct the refinement proofs manually.

For example, the layer  $L_1$  abstracts the array `page` into a Coq Map `st.page`, and transforms the memory operations `load_mem` and `store_mem`—offered by the bottom layer  $L_0$ 's machine model—into Map operations (`st.page#i` and `st.page#i<-s`) with boundary checks:

```

(* Low-level specifications *)
Definition get_page_low (i: nat) (st: ST) :=
  load_mem st ("page", i * 4) u32.
Definition set_page_low (i s: nat) (st: ST) :=
  store_mem st ("page", i * 4) s u32.
(* High-level specifications in L1 *)
Definition get_page_high (i: nat) (st: ST) :=
  if 0 <= i < MAX_PAGE then Some st.page#i else None.
Definition set_page_high (i s: nat) (st: ST) :=
  if 0 <= i < MAX_PAGE then Some st.page#i<-s else None.

```

Because of the data abstraction, the lifting refinement proof for layer  $L_1$  is not automated and has to be provided manually.

On the other hand, the layer  $L_2$  does not use data abstractions. For layer  $L_2$ , Spoq automatically generates the high-level specification of `alloc` from its low-level specification by applying a sequence of *transformation rules*, including unfolding definitions, merging near-duplicate sub-expressions, eliminating pre-determined branches and assertions, and performing mathematical simplification. The latter two rules are ap-

```

Fixpoint alloc_loop_high (r i: nat) (st: ST) :=
  match r with
  | 0 => (MAX_PAGE, st) (* no need of Some anymore *)
  | S r' => if st.page#i =? 0 then (i, st.page#i<-1)
    else alloc_loop_high r' (i + 1) st'
  end.

```

Figure 4: High-level specification for the alloc function.

plied by using the Z3 SMT solver [16]. For `alloc_loop_low` in Figure 3, Spoq first unfolds the definitions provided by  $L_1$  and simplifies the representation as shown below:

```

Fixpoint alloc_loop_low' (r i: nat) (st: ST) :=
  match r with
  | 0 => Some (MAX_PAGE, st)
  | S r' =>
    if 0 <= i < MAX_PAGE then (* <- always true *)
      if st.page#i =? 0 then
        if 0 <= i < MAX_PAGE then (* <- redundant *)
          Some (i, st.page#i<-1)
        else None
      else alloc_loop_low' r' (i + 1) st'
    else None
  end.

```

Spoq then applies rules to eliminate an inner `if` statement which is redundant and eliminate the outer `if` statement by inferring that `i` is always within the range, resulting in the high-level specification in  $L_2$  shown in Figure 4. Unlike the low-level specification, the high-level specification in  $L_2$  is self-contained and does not refer to anything from  $L_1$ . Thus, any modules depending on  $L_2$  can be reasoned about using  $L_2$  alone without the need to look at lower layers. Otherwise, after building dozens of layers, the specification at a higher layer may wrap many levels of definitions from various lower layers, making the verification non-modular and much harder.

Spoq automatically generates refinement proofs to verify the transformations that are applied to transform low-level into high-level specifications (Steps 5-6 in Figure 1). Since all specifications are guarded by machine-checkable proofs in Coq, there is no need to trust Spoq's specification generation algorithms or any Z3 results.

## 4 Generating Coq Representations

Spoq uses Clang to compile C code to LLVM IR, enabling it to support full C semantics and various extensions, including arbitrary type casting, integer-pointer conversion, inline assembly code, C macros that use GNU C extensions, and GNU C compiler directives. Spoq then translates LLVM IR code into an AST defined in Coq. IR code consists of structs, global variables, and functions. Spoq literally translates IR structs, similar to C structs, and global variables into their Coq representations, but does additional program reconstruction for IR functions. An IR function can be viewed as a control flow graph (CFG) over a set of basic blocks with an entry point. All instructions in a basic block are sequentially executed, and the last instruction either jumps to another block or returns from the function. Since systems code may contain goto statements and IR code is compiled with optimizations enabled, the CFG

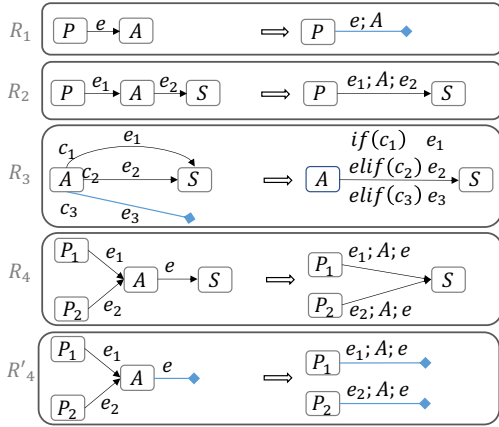


Figure 5: Rewrite rules for program CFGs without loops.

can be very complex and hard to reason about directly.

Spoq introduces a novel algorithm to merge each function’s CFG of basic blocks into one code block and reconstruct program structure using if-then-else, loop, continue, break, and return statements. Spoq only uses these statements to construct a program structure that is amenable to proof decomposition, which may not be the same as the program structure of the original source code. For example, any goto statements in the original source code will be eliminated. The algorithm reconstructs program structure by repeatedly applying a set of rewrite rules to reduce the size of the CFG by merging blocks and deleting edges. Spoq performs the reconstruction in IR. No attempt is made to reconstruct the original C code, which would bloat an otherwise minimal implementation.

**Reconstructing programs without loops.** For programs without loops, Spoq uses four rewrite rules to reconstruct programs from CFGs, shown in Figure 5. Each node denotes a code block and each edge denotes a change in control flow.  $A, P,$  and  $S$  in the nodes denote the instructions inside the respective blocks.  $c_1, c_2,$  and  $c_3$  at the beginning of edges denote the conditions to jump through the respective edges. Unlike regular CFGs,  $e, e_1,$  and  $e_2$  denote instructions attached to edges which will be executed when jumping through the respective edges. A blue edge ending with a rhombus denotes an edge without a destination, whose attached instructions must end with a continue, break, or return statement.

The CFG of a function without loops has no cycles, so Spoq can repeatedly apply the rewrite rules to reduce the graph to a single node. Rule  $R_1$  deletes a *dangling* node, a node with only one incoming edge  $e$  and no outgoing edge, and moves its instructions  $A$  to its incoming edge, which becomes an edge without a destination and has instructions “ $e; A$ .” Rule  $R_2$  deletes a *bridge* node  $A$ , a node with exactly one incoming edge  $e_1$  and one outgoing edge  $e_2$ , and redirects the incoming edge from its predecessor node  $P$  to its successor node  $S$  with instructions “ $e_1; A; e_2$ .” If all the outgoing edges of a node  $A$  either point to the same node  $S$  or do not have destinations, rule  $R_3$  merges all the edges into one

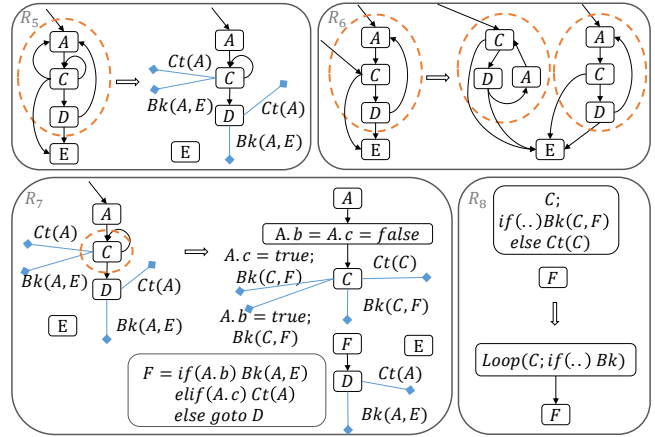
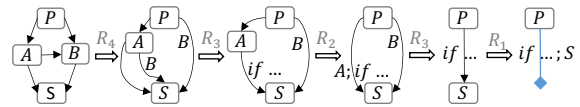


Figure 6: Rewrite rules for program CFGs with loops.

edge with branch statements. Since only the last instruction in a node changes the control flow, when a node has more than one outgoing edge, each edge must have a condition. If a node has multiple incoming edges but only one outgoing edge, rule  $R_4$  deletes the node and redirects all incoming edges to its successor node  $S$  with aggregated instructions. Rule  $R_4'$  is logically the same as  $R_4$ , but shows the case when the only outgoing edge does not have a destination.

The reconstruction algorithm prioritizes applying the first three rules and only applies  $R_4$  to the farthest valid node from the entry point if no other rules are applicable. We prove that this algorithm can rewrite any CFGs without loops into a single code block. The following example shows a sequence of rewrites to reconstruct the program structure from its CFG:



**Reconstructing programs with loops.** Loops introduce cycles into CFGs. For CFGs with cycles, Spoq computes the strongly connected components (SCCs). An SCC is the largest set of nodes in which every node is reachable from every other node. One node with self-pointed edges can also be an SCC. Spoq then uses four additional rewrite rules shown in Figure 6 to convert SCCs (marked by dotted orange circles) into loop-related statements.

Rule  $R_5$  breaks cycles in an SCC which only has one incoming edge (pointing to node  $A$  in the SCC), and all its outgoing edges point to the same destination (node  $E$  outside SCC). It redirects any edge to  $A$  in the SCC to having no destination, and appends  $Ct(A)$  (a continue statement for the loop  $A$ ) to the edge. It also redirects any edge to  $E$  in the SCC to having no destination, and appends  $Bk(A, E)$  (a break statement from the loop  $A$  to  $E$ ) to the edge. After the rewrite, there is no longer a cycle back to node  $A$  and the size of the SCC becomes smaller. When an SCC has incoming edges from more than one node, rule  $R_6$  duplicates the SCC for each node with incoming edges so that each SCC has only

one incoming edge. For nested loops in which the inner loop may directly jump out of the outer one, rule  $R_7$  converts such an SCC into one in which the jump target remains within the outer loop. Rule  $R_7$  inserts a new node  $F$ , and all outgoing edges from the inner loop are redirected via break statements to  $F$ . Flags are also appended to the outgoing edges. Node  $F$  contains instructions to jump to different destinations depending on the flag. Flag  $A.b$  means breaking the outer loop  $A$ ,  $A.c$  means going back to the beginning of the outer loop  $A$ , and no flag means breaking the inner loop. Once cycles are removed, rule  $R_8$  converts a node's instructions into a single `Loop` statement, and re-establishes the edge from the loop node to its successor indicated by the break statement.

**Assembly code.** Spoq also handles assembly code, representing assembly instructions as parameterized inductive types in Coq. Each instruction corresponds to one construct with the operand as the parameter. Since assembly is not a structured language, Spoq simply translates each assembly procedure or inline assembly statement into a list of assembly instructions in their Coq representation. For inline assembly, LLVM IR already encapsulates it as a function. Spoq extracts the assembly code into a separate assembly procedure, and replaces the original function body with a call to the assembly procedure, decoupling the inline assembly from the LLVM IR in the Coq representation. The current implementation only handles Armv8 assembly code.

**Semantics of Coq representations.** Once LLVM IR and assembly code is translated to its Coq representation, it can then be verified. This requires defining the semantics of LLVM IR and assembly instructions in Coq, to specify the behavior of the Coq representation. Semantics are defined with respect to a layer interface for a bottom layer machine model. The interface contains a machine state `st` and getter and setter methods that access objects in the machine state through object pointers. An object pointer is a pair  $(base, ofs)$ , where `base` specifies the object and `ofs` specifies the field or offset within the object. In other words, the semantics of LLVM IR and assembly instructions define how those instructions use the getter and setter methods and how they update the underlying machine state. The machine state contains memory blocks and registers, as discussed below.

LLVM IR semantics only depend on memory objects, each of which is a set of disjoint memory blocks that can be accessed using `load_mem` and `store_mem` methods through object pointers with boundary checks. A memory block is contiguous and its size is defined by the type of the respective structure or global variable. For example, the `page` array in Figure 2 is a memory block with  $(MAX\_PAGE \times 4)$  bytes and can be accessed using an object pointer  $(\text{"page"}, i)$ , where  $0 \leq i < MAX\_PAGE \times 4$ . The layer interface contains a variable environment providing a one-to-one mapping of variable names to corresponding addresses in memory.

For assembly code, Spoq models the semantics of the

Armv8 instructions based on not only memory block objects, but also register objects. For example, the register objects model that clearing the `VM` bit in `HCR_EL2` register will disable the stage-2 translation for EL1 and EL0. Since an assembly procedure is just a list of assembly instructions, the semantics of an assembly procedure is defined as applying the semantics for each assembly instruction in the list one after the other.

Based on CCALs, Spoq uses CPU-local reasoning and distinguishes memory objects as CPU-private memory, lock-synchronized memory, and lock-free memory. Each CPU-private memory object belongs to and can only be accessed by a particular CPU. Each lock-synchronized memory object is associated with a lock. When accessing a lock-synchronized memory object, Spoq checks that the corresponding lock is held by the local CPU. Accessing a lock-free memory object generates an event appended to a global log, and an event oracle is queried to simulate other CPUs' behavior before generating each event. Correct concurrent behavior is guaranteed in the same way as previous work using CCALs [38,40]. This event-based machine model assumes sequential consistency (SC). To propagate proof results for a system to Arm's relaxed memory hardware, users can follow the methods introduced by VRM [54] to verify that the system satisfies six weak-data-race-free conditions. This implies that the system exhibits no more behaviors when running on Arm relaxed memory hardware versus an SC model. Thus, any guarantees proven using the SC model still hold on Arm's relaxed memory hardware.

## 5 Synthesizing Identical Refinements

**Low-level specifications without loops.** Spoq recursively aggregates the small-step semantics of every IR statement in a function and generates a Coq definition to reflect the entire transition as the low-level specification of the function. Leveraging the reconstructed program structure, Spoq simply scans through the Coq AST representation, conducts case analysis starting with the first statement, and generates the corresponding Coq definition as a string based on the defined LLVM IR semantics. A small piece of Python pseudocode for assignment and branch statements is shown below:

```
def spec_gen (ast, spec):
    for n in range(len(ast)):
        i = ast[n]
        if isinstance(i, IAssign): # Assignment case
            s = f"let {coq_name(i.asg)} := {val(i.v)} in"
            spec.append(s)
        elif isinstance(i, IIf): # Branch case
            spec.append(f"if {coq_name(i.cond)} then")
            spec_gen(i.true_body + ast[n+1:], spec)
            spec.append(f"else")
            spec_gen(i.false_body + ast[n+1:], spec)
        ...
```

For an `IAssign` statement, which assigns a value to a temporary variable, Spoq generates a `let` binding in Coq. For an `IIf` statement, Spoq recursively invokes its specification generator `spec_gen` for each branch in the code and concatenates the branch body with the rest of the AST.



**Identical refinements without loops.** Spoq automatically generates identical refinement proofs by using a Coq tactic `lrefine`. The idea is to do case analysis for each conditional by recursively decomposing each conditional into two sub-proofs, one for when the conditional is true and another for when it is false. Once a branch body is reached with no further conditionals, the proof can simply show that if the low-level specification transforms the machine state from `st` to `st'`, then the small-step semantics of the Coq AST also transforms the machine state from `st` to `st'`. Spoq aggregates the sub-proofs for all the branch cases to form the overall refinement proof. Take the following pseudo-specification generated from an `if` statement as an example:

```
Definition foo_low (st: ST) :=
  if cond then foo_true_low st else foo_false_low st
```

The `lrefine` tactic will conduct case analysis over `cond`, which generates two sub-proof goals. The first goal is to prove that the AST transfers `st` to “`foo_true_low st`” with an additional hypothesis “`H0: cond = true.`” The `lrefine` tactic then executes the semantics of AST for one step by showing that the branch condition will be evaluated to `true` when `H0` holds and finally invokes `lrefine` recursively to prove that the first branch implementation will transfer `st` to “`foo_true_low st,`” a specification generated using the first branch. The second goal can be proved similarly.

**Low-level specifications for loops.** Spoq generates low-level specifications for loops using a recursive `Fixpoint` construction in Coq. A `Fixpoint` definition requires a decreasing argument, which has the type `nat` and decreases for each recursive call of the function. Spoq requires the user to provide a ranking function for each loop as the decreasing argument. It then generates low-level specifications for loops by filling in the parts marked with `{{ }}` in the template below:

```
1 Fixpoint _loop (n: nat) (bk rt: bool) {{Vi Vo}} st:=
2   match n with
3   | 0 => Some (bk, rt, {{Vo}}, st)
4   | S n' =>
5     match _loop n' bk rt {{Vi Vo}} st with
6     | Some (bk', rt', {{Vo'}}, st') =>
7       if bk' then Some (bk', rt', {{Vo'}}, st')
8       else if rt' then Some (bk', rt', {{Vo'}}, st')
9       else {{low-level spec of the loop body}}
10    | _ => None
11    end
12  end.
13 Definition _low {{args}} (st: ST):=
14   {{low-level spec before the loop}}
15   let n := {{rank i_Vi}} in
16   match _loop n false false {{i_Vi i_Vo}} st with
17   | Some (bk, rt, {{Vo}}, st') =>
18     if rt then Some ({{Vo}}, st')
19     else {{low-level spec after the loop}}
20   | _ => None
21   end.
```

For the loop, Spoq generates a `Fixpoint` construction such that one recursive call of the `Fixpoint` construction corresponds to one iteration of the loop, so its body is the low-level specification of the loop body (line 9). Five `Fixpoint` arguments track the state of the loop (line 1). `Vi` are

the input variables initialized before the loop and accessed by the loop body; they have initial values `i_Vi`. `Vo` are the output variables accessed after the loop that were also accessed in the loop body; they have initial values `i_Vo`. For example, the loop in `alloc` in Figure 2 simply has `i` for both `Vi` and `Vo`, with initial values `0` and `MAX_PAGE`, respectively. Spoq determines input and output variables and their initial values from syntactic analysis of the IR code. `n` is the decreasing argument, which is a natural number that is determined by the user-provided ranking function, which takes as input all the input variables of the loop. `n` is initialized using the ranking function over the initial value of input variables `i_Vi`, which sets the maximum number of “loop iterations” (line 15), and decreases by one for each “loop iteration” (line 4). Flags `bk` and `rt` indicate whether the loop has already been terminated by a `break` or `return` statement. The loop body (line 9) sets `bk` to true when executing a `break` statement or exiting when the loop condition becomes false, and sets `rt` to true when executing a `return` statement. `Fixpoint` will not make further changes once `bk` or `rt` is set to true (lines 7 and 8).

For the function containing the loop, Spoq generates low-level specifications for the code before the loop (line 14); invokes the `Fixpoint` with the initial values of the ranking function, flags, and variables (line 16); skips the rest of the function if `rt` is true (line 18); and generates low-level specifications for the code after the loop if not returned (line 19). Spoq will syntactically analyze the IR code and produce `Vi`, `Vo`, and their initial values `i_Vi` and `i_Vo`. Note that Figure 3 shows a simplified low-level specification that omits the `bk` and `rt` flags and uses a tail recursion style.

**Identical refinement proofs for loops.** Spoq proves identical refinements for loops using induction. The base case is trivial because the input machine states are the same. Spoq only needs to prove that the initial ranking function is non-negative. This is automated using a tactic `xlia`, extended from Coq’s tactic `lia`, a decision procedure for arithmetic. The induction step is to show that when the input machine states for the low-level specification and Coq AST are the same after the  $i$ -th iteration and both `bk` and `rt` are false, the output machine states are still the same after the  $(i+1)$ -st iteration. The  $(i+1)$ -st iteration may have one of three outcomes: 1) continue to the next iteration, 2) break the loop due to a `break` statement or the loop condition becoming false), and 3) return from the function. For all three outcomes, Spoq first proves that the loop body and `Fixpoint` body have the same semantics by recursively invoking `lrefine`. Spoq then proves additional properties for each outcome. For the first outcome, Spoq proves that the ranking function decreases by at least one and is still greater than zero using `xlia`. This guarantees that the loop must terminate after at most the number of iterations indicated by the initial ranking function. For the second outcome, Spoq proves that `bk` is true after the iteration, and the ranking function is still non-negative when the loop condition becomes false using `xlia`. For the third

outcome, Spoq proves that `rt` is true after the iteration. Note that the `Fixpoint` function continues the iteration after `bk` or `rt` is true but will not make any changes to the state.

Spoq automatically generates the identical refinement proof for a loop if the loop is not contained within a conditional in the function. However, if the loop is contained within a conditional, or a series of conditionals, this results in the loop being used in multiple branches of execution, which Spoq currently does not automatically handle. In this case, the user will see that the loop termination proof failed in one or more branches, and needs to copy and paste the induction proof template into the other branches of execution with possible minor modifications; this is generally straightforward to do.

**Assembly code.** Spoq generates low-level specifications for assembly code by evaluating the assembly instruction list. The current implementation only supports automatic generation of low-level specifications for assembly code without jumps. Spoq simply evaluates instructions sequentially and outputs the machine state of the last instruction. If the destination of a call instruction is a C function, Spoq uses registers according to the Procedure Call Standard for the Arm 64-bit Architecture (AAPCS64) [5]. Spoq sets the arguments to the values in the argument registers according to AAPCS64. After the function call, Spoq checks the linker register of the machine state and evaluates the assembly instruction from where the linker register points. After returning from the function call to assembly code, Spoq sets the value of the caller-saved registers to `UNKNOWN` because the caller cannot assume any value in the caller-saved registers according to AAPCS64. Spoq disallows reads from any register with value `UNKNOWN`; assembly code must write to the caller-saved register first before it can be read. This helps prevent unexpected information leakage from registers.

By using the AAPCS64 calling conventions for assembly code functions so that arguments and return values are treated the same as C code functions, Spoq provides a unified approach to generating low-level specifications for assembly and C code. This includes using the same type `value` used in the IR semantics for assembly code. This unified approach makes it possible to link the proofs for assembly and C code.

Spoq generates low-level specifications for inline assembly in the same manner as other assembly code, since it already extracts the inline assembly into a separate assembly code procedure. However, Spoq requires that the operands used in inline assembly are C variables specified in the input or output operand list, system registers, and constants. Directly reading or writing general-purpose registers is disallowed to ensure proof correctness when linking inline assembly and C code, as the compiler may use them for temporary variables [40].

Spoq automatically generates identical refinement proofs for assembly code, which is straightforward without jumps as there are also no loops. The proof simply shows that the low-level specification and assembly instruction list transform the machine state in the same way.

## 6 Synthesizing Lifting Refinements

**High-level specifications.** Spoq generates high-level specifications by applying a set of transformation rules to low-level specifications to make them self-contained and simple. Spoq uses 12 transformation rules shown in Figure 7, though additional rules can easily be added. Spoq uses the Z3 SMT solver to apply rules involving symbolic execution or mathematical simplification. The goal of the transformation rules is to simplify the required control flow and eliminate as much as possible unnecessary operations.

$T_1$  unfolds a function's definition in an expression. Functions defined in lower layers that are called in the low-level specification are generally unfolded as part of the high-level specification to make it self-contained. Unfolding may also provide opportunities to apply other transformation rules to eliminate unnecessary operations to further simplify the specification.  $T_2$  eliminates a `let` assignment by substituting the variable with its value, which helps find opportunities for simplifying expressions.  $T_3$  eliminates an `if` branch if both branches are the same.  $T_4$  eliminates a `match` statement by syntactically determining which pattern matches the source value.  $T_5$  eliminates a `match` statement if both the source and return values are of `Option` type, and if the source value is `None`, the return value is `None`. It eliminates the `match` by making `body` the return value for all source values that are not `None`.  $T_6$  transforms a `match` statement in which the source value matches the pattern and is used in the return value by substituting the pattern in the return value. This can provide more opportunities for simplification since patterns are more specific.  $T_7$  moves the control flow of the source value to the outside of the `match` statement. Spoq tries to simplify the source value of `match` statements to make it easier to determine matching patterns.  $T_8$  moves the control flow within an expression to the outside of the expression to aggregate computations within the expression, which helps find opportunities for simplifying expressions.  $T_9$  does various simplifications for getter and setter methods. Here  $i$  and  $j$  indicate different fields. Whether  $i$  equals  $j$  can be determined syntactically (if they are structure names), or by Z3 (if they are integer indices).  $T_{10}$  performs symbolic execution using Z3 to identify whether the assertion of a `rely` is valid or invalid. If the assertion is always true, then `rely` is redundant and can be removed. If the assertion is always false, the statement can simply return `None`.  $T_{10}$  will do nothing if Z3 cannot decide if the assertion is true or false.  $T_{11}$  performs symbolic execution using Z3 to simplify `if` statements.  $T_{12}$  simplifies math expressions using Z3. For example, Spoq applies  $T_1$ ,  $T_2$ , and  $T_{11}$  to generate the high-level specification in Figure 4 from its low-level specification.

While the transformation rules can be applied in different orders to yield the same result, the order in which the rules are applied can have a significant impact on the execution time required. Spoq reduces execution time by applying the rules in stages. In the first stage, it applies rules  $T_2 - T_8$  and the  $T_9$

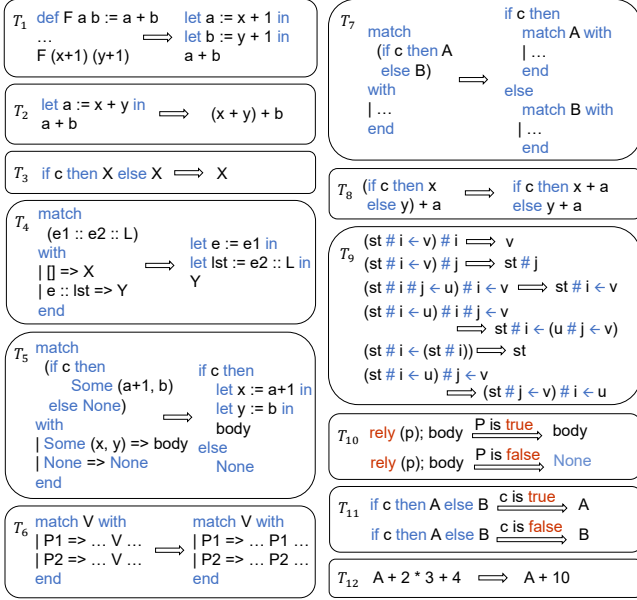


Figure 7: Transformation rules for high-level specifications.

syntactic transformations. In the second stage, it applies rule  $T_1$ , then repeats applying the rules from the first stage. Spoq unfolds only one function, and only when no other syntactic rules can apply, because unfolding multiple functions too early can cause extra work. In the extreme case, unfolding all functions first will cause the size of the specification to explode and result in many unnecessary tests on each expression in each unfolded function body. In the third stage, it applies rules that use Z3, specifically rules  $T_9 - T_{12}$ , then repeats applying the rules from the first and second stages. Spoq applies syntactic rules first to simplify the specification as much as possible before applying Z3 rules because Z3 rules take much longer to process. To avoid long Z3 processing times, Spoq enforces a short timeout on Z3 operations, which is set to half a second by default. Essentially, Spoq repeatedly applies all rules until the high-level specification converges, meaning the rules no longer change the specification.

Using transformation rules to make the high-level specification of each layer self-contained generally results in the high-level specification being of larger size than its corresponding low-level specification. However, this size increase is outweighed by the ability to use the self-contained specification to simplify reasoning for higher layers, especially with regard to reasoning about higher-level properties based on the top layer high-level specification.

**Lifting refinement proofs.** Spoq automatically generates lifting refinement proofs to prove that the low-level specification refines the high-level specification generated by the transformation rules. This will necessarily be the case for transformation rules done in Coq, so the task reduces to reconstructing the proofs in Coq for all transformations done by Z3; there is no need to trust any results from Z3. Spoq uses a Coq tactic library to enable the proof automation.

Spoq simplifies the construction of refinement proofs by introducing annotated high-level specifications, which are the same as high-level specifications except that they have additional annotations that encapsulate the results of all of the Z3 transformations applied. For example, if  $T_{11}$  is applied, there will be an annotation showing that  $A + 2 * 3 + 4 = A + 10$ , which serves as a hint for constructing proofs. Spoq generates the annotations as it is generating the high-level specification. Spoq then uses the annotations to tell Coq what step-by-step syntactic substitutions it should perform to prove the low-level specification refines the annotated high-level specification. Because the annotations tell Spoq what transformations to do, it only has to validate them in Coq, which is much easier than automatically discovering the transformations in Coq; that would be difficult without Z3. Spoq finally trivially proves that the annotated high-level specification refines the high-level specification by showing that removing the annotations does not change the machine behavior. The two-part refinement proof shows that the low-level specification refines the high-level specification.

Spoq introduces a Coq tactic `hrefine` to automate the core part of the proof, namely proving that the low-level specification is equivalent to the annotated high-level specification. The strategy of `hrefine` is similar to the one for `lrefine` used for the identical refinement proof discussed in Section 5. The `hrefine` tactic analyzes the structure of the annotated high-level specification, decomposes it into all possible branches of state transitions, and conducts the proof for each branch. For each branch, all `match`, `if`, and `rely` are eliminated because the branch corresponds to a specific set of values for their conditions. Each branch therefore has a list of conditions and annotations. Spoq uses those conditions and annotations to simplify the low-level specification and prove that the low-level specification has the same behavior as the high-level one for that branch. It then repeats this process to prove the refinement for each branch.

Section 7 shows that Spoq was able to automatically generate all lifting refinement proofs involving Z3 transformations in verifying a multiprocessor KVM hypervisor. However, it is theoretically possible for there to be Z3 transformations for which Spoq is not able to generate lifting refinement proofs, in which case the user needs to manually complete those proofs.

Spoq also uses Coq tactics to automatically generate lifting refinement proofs for `Fixpoint` constructions, which are used in high-level and low-level specifications for functions with loops. The proofs use induction and are straightforward to generate because they only involve `Fixpoint` constructions, which are guaranteed to terminate. The hard part of refining loops to `Fixpoint` constructions and completing termination proofs has already been done in the low-level specifications.

Using Spoq provides significant advantages in terms of proof modularity over previous approaches that required users to manually write high-level specifications and proofs [20, 38, 40]. Because creating a self-contained



high-level specification often involves unfolding function definitions from lower layers, any change to an implementation at a lower layer can require rewriting the high-level specifications for all higher layers, which also requires rewriting their refinement proofs. This makes it difficult to port specifications and proofs as a software implementation evolves over time if high-level specifications and refinement proofs are manually written, as many of them may have to be manually rewritten. With Spoq, the impact of an implementation change can be localized to its respective layer, even if that layer requires writing high-level specifications or proofs manually, since high-level specifications and proofs for higher layers can be automatically generated. This makes it much easier to port specifications and proofs across software updates.

**Assembly code.** Spoq generates high-level specifications and lifting refinement proofs for assembly code without jumps in the same manner as for C code. The current implementation leaves it to the user to write specifications and refinement proofs for assembly code with jumps.

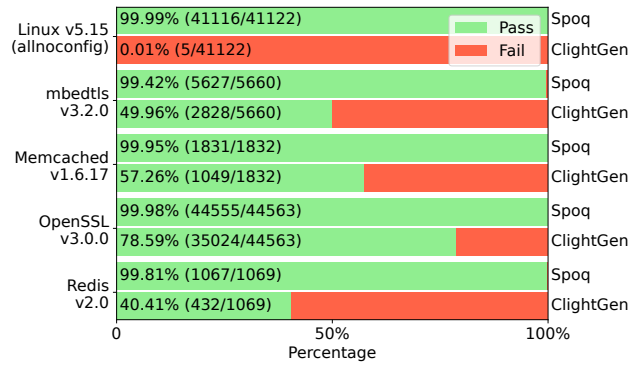
## 7 Evaluation

We have implemented a Spoq prototype, which consists of three components: the translator from systems code into Coq, the specification and proof generator, and the Coq libraries for LLVM IR and assembly semantics and tactics. The three components are implemented using 4K lines of code (LoC) in C++ and Python, 6K LoC in Python, and 5K LoC in Coq, respectively. We evaluated Spoq’s effectiveness in translating C systems code into Coq for various widely used open-source software, and verifying a KVM hypervisor implementation.

### 7.1 Translating system software into Coq

Since the first step in verification is to translate systems code into Coq, we evaluated Spoq’s ability to do so for the applications, libraries, and Linux kernel version listed in Figure 8. We used the Makefile for the source code tree of each application, library, and kernel to build the source code using the default configuration, but output LLVM IR (.ll) files, in some cases by modifying the Makefiles by replacing the `-o` compilation option to output an executable with the `-S -emit-llvm` option to output LLVM IR files, which are then read by Spoq to translate them into Coq. The Linux kernel uses a more complex KBuild system [29], but no modifications were needed since it already accepts the `-S -emit-llvm` option to output LLVM IR files.

For comparison, we also tried to use ClightGen to translate the systems code into Coq. This required much more effort to the build source code trees because many of the compiler flags are not accepted by ClightGen. Instead, for most cases, we ran the existing Makefiles to get the compilation commands executed and saved them to a file, then used a



**Figure 8:** Translating C code into Coq. Each bar shows how many of the total number of C functions are successfully translated.

script to filter options not supported by ClightGen, then reran the filtered compilation commands using ClightGen instead.

Figure 8 shows the results for translating C systems code into Coq using Spoq versus CompCert’s ClightGen. Across all of the applications, libraries, and the Linux kernel, Spoq successfully translates over 99% of the functions in the source code into their Coq representations. The failures were caused by currently unsupported LLVM instructions, mainly advanced branching instructions (e.g. `callbr`, `invoke`, `resume`). Support for them is left for future work.

Spoq performs significantly better than ClightGen, which fails almost entirely on the Linux kernel and only translates roughly 50% of the functions in the source code into their Coq representations for most cases. Its best performance is on OpenSSL, for which it is still able to only translate less than 80% of the functions in the source code into Coq representations. ClightGen fails due to numerous unsupported C features, including variable-sized arrays, function parameters or return values with `union/struct`, additional keywords, C statements, and other unsupported inline assembly features. Furthermore, for the Linux kernel, GNU C directives are ubiquitous in almost all header files included by source code files and prevent ClightGen from translating the kernel source code into Coq.

Not only does Spoq perform far better than ClightGen in translating systems code into Coq representation, but it has a much smaller implementation. The module in Spoq responsible for translating systems code into Coq consists of 2.7K LoC in Python and 1.3K LoC in C++, the latter to make use of the official LLVM library to parse LLVM IR files. Its minimal implementation avoids bloating the TCB. In contrast, ClightGen is enormous, consisting of at least tens of thousands of lines of unverified OCaml code. ClightGen performs worse than Spoq and increases the TCB size much more significantly than Spoq as well.

### 7.2 Verifying a KVM hypervisor

We evaluated Spoq’s ability to reduce proof costs by verifying SeKVM, a retrofitted version of the KVM/Arm



```
#define __hyp_text __section(.hyp.text) notrace
u32 __hyp_text mem_region_search(u64 addr)
```

(a) Unsupported compiler directive.

```
/* Original source code:
 * inline assembly and macro of a C statement */
u32 __raw_readl(const volatile void __iomem *addr){
  u32 val;
  asm volatile("ldr %w0, [%1]\r\nldar %w0, [%1]",)
  : "=r" (val) : "r" (addr));
  return val;}
#define readl_relaxed(c) \
({ u32 __r = \
  le32_to_cpu((__force __le32)__raw_readl(c)); \
  __r;})

/* Verified source code:
 * original source code replaced with only C function
 * declaration so it can be parsed by ClightGen. */
u32 readl_relaxed(u64 addr);

(* Specification modeling the behavior of
 * readl_relaxed; implementation unverified. *)
Definition readl_relaxed_spec (addr: Z) (st: ST) :=
  (ZMap.get st.(mem) addr, st).
```

(b) Unsupported GNU Inline Assembly and C statement.

**Figure 9:** Example SeKVM changes required to use ClightGen.

hypervisor [13–15, 37] that was previously verified in Coq [38, 39, 54]; only its trusted core needed to be verified to guarantee the security properties of the entire multiprocessor hypervisor. We updated SeKVM to run on additional hardware, specifically the Raspberry Pi 4, which involved modest changes to its previously verified codebase. However, this required updating the proofs, so we used Spoq to verify the updated version, and compare the proof effort to the manually written Coq proofs for the earlier version of SeKVM.

**Generating Coq representations.** We first used Spoq to automatically translate the source code of the trusted core of the updated hypervisor version into Coq. Spoq successfully translated all of the 3.8K LoC of C and Arm assembly code into Coq. The same code that is compiled to execute is used for verification; there is no difference, ensuring that the proofs hold at the source code level for the code that is executed. This is in contrast to the previous work to verify SeKVM, which used ClightGen to translate its implementation into Coq. This required further retrofitting of the source code because of its use of many features unsupported by ClightGen, including removing all header files with versions that were amenable to translation by ClightGen.

Figure 9 shows examples of the retrofitting required to use ClightGen. Figure 9a shows a GNU C compiler directive `__section` which tells the linker to link the function into a special text section that SeKVM later isolates and protects from the rest of the kernel. ClightGen does not support such GNU C compiler directives, which are heavily used in systems code to control compilation and linking behavior. To use ClightGen, we first need to remove those GNU C compiler directives from all functions. Figure 9b shows a C macro `readl_relaxed` with inline assembly. ClightGen does not support such C macros or inline assembly. To use ClightGen, we need to either rewrite

all such macros into standard C functions, or model them as abstract functions whose implementations are not verified and must be included in the TCB. Figure 9b shows the latter approach. The macro is replaced with just a function declaration so it can be translated by ClightGen, and a specification is written for the function, but the function implementation cannot be verified. There are over a hundred such functions in the original source code. These required changes result in a gap between the code that is verified versus the code that is compiled and executed. Unfortunately, without supporting features such as GNU C compiler directives, the verified code cannot be directly compiled and executed.

**Generating specifications and proofs.** We then used Spoq to generate the top-level specification for SeKVM, including all layer specifications and refinement proofs. Table 1 shows the manual proof effort required to verify SeKVM’s functional correctness using Spoq, as measured by the LoC in Coq that still needed to be manually written to complete the verification. We also propagated the proofs to Arm’s relaxed memory hardware, but omit details as it is similar to VRM’s proof [54].

We wrote less than 100 LoC to provide the layer structure in a layer configuration file consisting of the same 34 layers as the original proofs for SeKVM; the changes in the updated version of SeKVM were minor enough that no changes in the layer structure were needed. We wrote 0.5K LoC for the bottom layer machine model for concurrency-related structures.

For C code without loops and Arm assembly code without jumps, Spoq automatically generated all low-level specifications and identical refinement proofs. For C code with loops, Spoq automatically generated all low-level specifications given a ranking function for each loop, each requiring 2 LoC. For C code with loops within conditionals, we wrote 0.8K LoC for identical refinement proofs that could not be automated by the current Spoq prototype, much of which involved copying and pasting of Coq code for termination proofs when multiple conditional branches used the same loop.

For C code and Arm assembly code without jumps, Spoq automatically generated all high-level specifications and lifting refinement proofs that do not use data abstractions. No manual proofs were required to verify Z3 transformations. For assembly code with jumps, we wrote 0.3K LoC for specifications and 0.1K LoC for refinement proofs, without decomposing specifications and proofs into low-level and high-level ones. For layers using data abstractions, one for locks and three for page tables, we manually wrote high-level specifications and lifting refinement proofs. For high-level specifications, we wrote 1.0K LoC for layers using data abstractions. For lifting refinement proofs, we wrote 0.8K LoC for locks, 2.5K LoC to show multi-level page tables refine a single-level page mapping, and 0.9K LoC to show data structures tracking ownership of physical pages refine an abstract map.

**Reducing manual proof effort.** Table 1 compares the proof effort to verify SeKVM using Spoq versus the manually writ-

LoC in Coq	Original	Spoq	Reduction
Layer configuration	—	0.1K	—
Machine model	1.8K	0.5K	72%
Low-level specifications for C	5.6K	0	100%
Ranking function	—	26	—
High-level specifications for C	5.5K	1.0K	82%
Specifications for Asm	0.5K	0.3K	40%
Identical refinement proofs for C	3.6K	0.8K	78%
Lifting refinement proofs for C	14.7K	4.2K	71%
Refinement proofs for Asm	1.8K	0.1K	94%
Security proof	4.8K	3.0K	38%
<b>Total for functional correctness</b>	<b>33.5K</b>	<b>7.0K</b>	<b>79%</b>
<b>Total w/security</b>	<b>38.3K</b>	<b>10.0K</b>	<b>74%</b>

**Table 1:** Manual proof effort to verify SeKVM.

ten proofs for the original version of SeKVM. The original manual proof effort required writing more than 3 times as many lines of specification and 5 times as many lines of proof as verified source code. Spoq only required writing a third as many lines of specification and roughly 1.4 times as many lines of proof as verified source code. In terms of LoC, Spoq reduced the overall manual proof effort by more than 70% compared to the original manually written proofs. The largest reductions in proof effort were for writing the specifications themselves. Spoq reduced manual effort for writing specifications by more than 90% overall, including eliminating the cost for specifications without data abstractions. Spoq reduced manual effort for refinement proofs by more than 70% overall, including eliminating the cost for C code without loops or data abstractions. Spoq reduced manual effort for refinement proofs for assembly code by more than 90% and linked them together with the proofs for C code, in contrast to the original assembly code proofs for SeKVM. Spoq largely eliminated the cost of using intermediate layers to modularize proofs, a substantial cost in the original manually written proofs, as the vast majority of those layer specifications and refinement proofs were automatically generated by Spoq.

Spoq also reduced the manual effort in defining the bottom layer machine model by roughly 70% due to three reasons. First, Spoq automatically derived many aspects of the abstract machine model from the source code. In contrast, the machine model for the original manually written proofs did not have such a correspondence with the source code and had to be manually written. Second, Spoq can use a simpler machine model because it does not need data oracles [38], which were introduced in the original manually written proofs to verify security properties. We discuss below how we verify security properties in a different manner, making data oracles unnecessary. Finally, Spoq does not need to include various getter and setter functions in the bottom layer, which were required in the original manually written proofs. These getters and setters, written using various Linux macros, previously had to be manually specified as part of the bottom layer specification because they could not be translated by ClightGen into Coq and hence could not be verified. In contrast, Spoq automati-

cally translated these getters and setters into Coq and verified them, eliminating them from the bottom layer specification.

We compared the Coq code generated by Spoq versus the original manually written proofs for SeKVM to provide a measure of the quality of the generated specifications and proofs versus what would be produced by humans. Spoq generated 2.5K, 6.6K, 4.2K, 6.9K, and 17.5K LoC in Coq for the machine model, low-level specifications for C code, high-level specifications for C code, identical refinement proofs for C code, and lifting refinement proofs for C code, respectively. In most cases, the generated Coq code was only modestly larger than what was produced by a human writing hand-tuned Coq specifications and proofs. In fact, Spoq generated tighter high-level intermediate layer specifications than the original manually written specifications. The top-level specification generated by Spoq was 1.6K LoC in Coq. This is essentially the same size as the original manually written top-level specification, though it is quite different as it is based on a different machine model derived from the source code for the bottom layer. The quality and complexity of the top-level specification is especially important since it should be simple enough that it can be used to prove higher-level properties of the system.

**Proving security properties.** To demonstrate the usefulness and correctness of the top-level specification generated by Spoq, we used it to verify the security properties of SeKVM, specifically that it protects the confidentiality and integrity of virtual machine (VM) data. The original manually written proofs used noninterference to prove the security properties along with data oracles for declassification. We instead leverage the ideal/real paradigm to prove security properties, introduced in our recent work on verifying the firmware for the Arm Confidential Compute Architecture [40]. We define an ideal machine model that guarantees the security of each VM’s private data regardless of the behavior of the hypervisor. The ideal machine defines for each VM a logically isolated memory space and register set, and directs all memory and register accesses from VMs to the logical state unless data declassification is defined. To account for data declassification in SeKVM in which a VM can make requests to dynamically start and stop sharing a piece of memory with the hypervisor, the ideal machine moves data from the VM’s logical memory to shared memory and vice versa. The VM accesses its private data from its logical memory space, and accesses the shared data from the shared physical memory. By definition, the per-VM isolated state is only accessible by the VM itself, so the confidentiality and integrity of VM data is naturally guaranteed in the ideal machine. We then prove the top-level specification refines the ideal machine, which verifies that SeKVM indeed protects the confidentiality and integrity of VM data.

The security proof provides three key advantages compared to the original security proofs based on noninterference. First, it does not require incorporating data oracles in the machine model and in various layer specifications, decoupling the security proof from verifying functional correctness. Second,

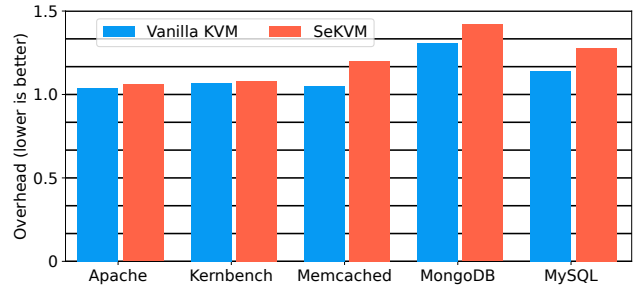
Name	Description
Apache	Apache server v2.4.41 handling 100 concurrent requests via TLS/SSL from remote ApacheBench [2] v2.3 client, serving index.html of the GCC 7.5 manual.
Kernbench	Compilation of the Linux kernel v4.18 using allnoconfig for Arm with GCC 9.3.0.
Memcached	Memcached v1.5.22 handling requests from a remote memtier [49] v1.2.11 client with default parameters.
MongoDB	MongoDB server v3.6.8 handling requests from a remote YCSB [10] v0.17.0 client running workload A with 16 concurrent threads and operationcount=500000.
MySQL	MySQL v8.0.31 running sysbench v1.0.11 with 32 concurrent threads and TLS encryption.

**Table 2:** Application benchmarks.

the proof itself is simpler, reducing manual proof effort. Table 1 shows the manual proof effort for the security proof using this approach is 38% less than the original security proof using noninterference, though this reduction in proof effort is unrelated to using Spoq. Finally, and most importantly, the security proof only needs to trust the specification of a small idealized secure machine model, which is roughly 200 LoC; the much larger specification of the real system does not need to be trusted. The trusted specification defines how VMs load and store private data to their logical isolated space and specifies the data declassification policy for moving data between the logical isolated and shared machine states.

**Performance of verified implementation.** We directly compiled the Spoq-verified SeKVM source code into a binary image, and executed it on a Raspberry Pi 4B with 8 GB RAM, 64 GB SanDisk SD card, and a built-in 1 Gbps NIC. We measured its performance by running the application workloads listed in Table 2 in a VM using SeKVM. For comparison, we also ran the workloads in a VM using vanilla KVM and natively on the hardware. Each VM was configured with 2 vCPUs and 4 GB RAM. vCPUs were pinned to individual physical cores, VHOST networking was used, and virtual block storage devices were configured with `cache=none` [12, 24, 33, 52]. When running natively, we restricted the workloads to use 2 CPUs and 4 GB RAM to provide a fair comparison. VMs used a vanilla Linux v5.4 kernel as their guest OS. The VM on SeKVM included modified virtio drivers in its guest OS to support SeKVM. The Raspberry Pi ran a proprietary Linux v5.4.55 kernel [45]. It lacks support for virtio front-end drivers so could not be used as a guest OS. For client-server applications, clients ran on an x86 machine with 10-core Intel Xeon CPU E5-2640 2.4 Ghz CPU, 48 GB RAM and a NetXtreme BCM5719 1 Gbps NIC, connected to the Raspberry Pi via a Netgear GS308 1 Gbps switch.

Figure 10 shows application workload performance when using VMs with vanilla KVM and SeKVM. Performance was normalized to native execution; lower is better. The performance results are consistent with those previously reported for SeKVM [54], with worst case overhead being



**Figure 10:** Application benchmark performance.

less than 15% compared to vanilla KVM. I/O intensive application workloads incurred higher overhead because the hypervisor cannot access VM memory unless the virtio front-end driver makes explicit hypercalls to request memory pages used for I/O be temporarily accessible to the hypervisor to pass the I/O data to the back-end driver in the host.

## 8 Limitations

Spoq’s TCB includes the Clang and LLVM toolchains, Spoq’s translator, and Spoq’s semantic definitions for LLVM IR and assembly. The translator is currently unverified and supports a subset of LLVR IR and Arm assembly, so it may fail to translate some source code into Coq. Spoq’s specification and proof generator are not part of its TCB. Enhancing their support for assembly code with jumps is an area of future work.

The Z3 solver is currently the bottleneck in Spoq’s runtime performance. Synthesizing high-level specifications for relatively large functions can take over 30 minutes because it may involve thousands of Z3 queries. Nevertheless, automatically generating specifications and proofs for SeKVM only takes two hours on an AWS machine with an 8-core 2.3 GHz Intel Xeon CPU E5-2686 v4 and 32 GB RAM, an insignificant amount of time compared to the time it takes to manually write specifications and proofs.

Spoq currently relies on users to complete all data abstraction proofs. Developing a library of commonly used data abstraction proofs for proof automation is an area of future work.

## 9 Related Work

**Verified systems in C.** seL4 [31] presents the first machine-checked functional correctness proof of an OS kernel. It used an unverified parser to translate C into Isabelle/HOL, and is manually proved with simplified C semantics. For example, pointers to local variables are disallowed by the simplified C semantics. Assembly code is also unverified. AtomFS [61] used a verification framework [56] that does not support assembly code or full C semantics. Many verified systems [3, 8, 11, 20, 30, 32, 38–41, 54] used ClightGen. For code that can be parsed by ClightGen and compiled by CompCert, the CompCert toolchain can guarantee proofs hold at the assembly level. However, CompCert cannot make



any guarantees regarding concurrent code even if it can compile, and our results show that ClightGen cannot support verification of most real-world unmodified systems code.

**Modeling and verifying LLVM IR.** VeLLVM [59] includes formal semantics and tools to verify LLVM IR code in Coq. VeLLVM adopts CompCert’s sequential machine memory model, so it cannot verify concurrent systems. It directly models small-step semantics of IR instructions in CFGs, making it problematic to use for systems with complex control flows. CreLLVM [28] extends VeLLVM to verify compiler optimization passes, but shares the same limitations of VeLLVM. VIR [48] also models small-step semantics of IR instructions in CFGs, suffering the same problems as VeLLVM. K-LLVM [36] defines LLVM IR operational semantics in the K framework, but cannot be used for deductive reasoning. SeaHorn [22] statically checks assertions in C programs by translating them to LLVM IR, then using the IR with an SMT solver and abstraction interpretation. Such automated verification tools cannot verify the functional correctness of a complex system. It is difficult to define program specifications using just assertions, and SMT solvers and abstract interpretation cannot prove complex proof goals.

**Automating systems verification.** AutoCorres [17] synthesizes specifications from C programs based on a fixed and simplistic machine model, which cannot be used to verify concurrent systems. It only supports an even more limited subset of C than ClightGen and does not support assembly code. The specifications generated are low level yet machine dependent, making them difficult to use to verify higher-level properties.

Push-button verification is a fully automated verification technique that has been used to verifying a file system [50], compiler [53], and OS kernel [43, 44, 51]. Users only need to write specifications in addition to the system implementation, and the verification framework automatically completes the proofs. However, implementations have restrictive constraints, such as uniprocessor only and constant bounds for loops so SMT solvers can be used. Unlike Spoq, verification requires manually defined specifications, does not hold for concurrent systems, and lacks machine-checkable proofs, as the unverified SMT solver provides no proof of its answer or any way to express a proof that can be machine checked.

Verification-aware programming languages such as Dafny [34] and F\* [47] have been used to implement verified storage systems [25, 26] and crypto libraries [46, 60]. Developers write code, specifications, and proofs all together in the same language. A compiler validates users’ proofs, in part using an SMT solver, and generates source code in familiar programming languages, which can in turn be further compiled and executed. Building on Dafny, Armada [42] uses a set of pre-built proof strategies to generate refinement proofs between levels of intermediate specifications, which users are expected to write to bridge the semantic gap between an implementation and its high-level specification. Unlike

Spoq, layers are not supported and systems written in C and assembly code cannot be verified without being rewritten.

**Decompilation.** Decompilation techniques recover a program’s source code given only its binary [1, 6, 9, 23, 27, 57, 58], though the recovered and original source code generally do not match. Some techniques do not reconstruct program structure [1, 58], some do so with goto statements [6], and some only do so with various restrictions on program CFGs [57]. More recent work can reconstruct program structure for arbitrary CFGs [23] without using goto statements, but requires a much more complex algorithm than used by Spoq. In contrast, Spoq employs a simpler algorithm to reconstruct program structure for arbitrary CFGs, and keeps the original LLVM IR instructions, which are much simpler and more rigorously defined than C, instead of trying to recover source code. To support proof decomposition and simplify specification synthesis, Spoq intentionally does not employ a richer variety of source code primitives such as goto or switch statements. Its resulting representation is more amenable to verification. Its design and implementation is far simpler than previous approaches to keep its TCB small, which is important for verification.

## 10 Conclusions

Spoq is the first system that can automate the generation of Coq representations, specifications, and proofs for C systems code to enable machine-checkable verification of concurrent system software. Spoq translates C systems code compiled into LLVM IR directly into Coq, converting IR control flow graphs into structured program functions to simplify verification while supporting full C semantics, including GNU C extensions and inline assembly. Using a layering proof strategy, Spoq introduces novel Coq tactics and transformation rules to automatically synthesize layer specifications and refinement proofs, even for functions with loops. Users can interact with Spoq to further refine the generated specifications and proofs at any layer. We used Spoq on commodity system software, such as the Linux kernel, to translate over 99% of their source code directly into Coq for verification. We also used Spoq to verify a multiprocessor KVM hypervisor implementation, showing that it reduces manual proof effort by over 70% while ensuring that the proofs hold for the unmodified implementation that is compiled and executed.

## 11 Acknowledgments

Jay Lorch provided helpful and meticulous feedback on earlier drafts. This work was supported in part by three Amazon Research Awards, a Guggenheim Fellowship, a VMware Systems Research Award, an NSF CAREER Award, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the founder of and has an equity interest in CertiK.



## References

- [1] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 2016)*, pages 583–600, Austin, TX, August 2016.
- [2] Apache Software Foundation. ab - Apache HTTP Server Benchmarking Tool. <http://httpd.apache.org/docs/2.4/programs/ab.html>. Accessed on December 13, 2022.
- [3] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*, pages 1–17, Saarbrücken, Germany, March 2011.
- [4] ARM Ltd. ARM Architecture Reference Manual ARMv8-A DDI0487A.a, September 2013.
- [5] ARM Ltd. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64). <https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs64.pdf>, April 2022.
- [6] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013)*, pages 353–368, Washington, D.C., August 2013.
- [7] Edouard Bugnion, Jason Nieh, and Dan Tsafir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.
- [8] Hao Chen, Xiongnan Newman Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 431–447, June 2016.
- [9] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.
- [10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 143–154, Indianapolis, IN, June 2010.
- [11] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664, June 2016.
- [12] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.
- [13] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.
- [14] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.
- [15] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.
- [16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2008)*, pages 337–340, Budapest, Hungary, March 2008.
- [17] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don’t Sweat the Small Stuff: Formal Verification of C Code without the Pain. *ACM SIGPLAN Notices*, 49(6):429–439, June 2014.
- [18] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.
- [19] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Newman Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, October 2019.
- [20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating*

- Systems Design and Implementation (OSDI 2016)*, pages 6530–669, Savannah, GA, November 2016.
- [21] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.
- [22] Arie Gurfinkel, Temesghen Kahsay, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*, pages 343–361, San Francisco, CA, July 2015.
- [23] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A Comb for Decompiled C Code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS 2020)*, pages 637–651, Taipei, Taiwan, October 2020.
- [24] Stefan Hajnoczi. An Updated Overview of the QEMU Storage Stack. In *LinuxCon Japan 2011*, Yokohama, Japan, June 2011.
- [25] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 99–115, November 2020.
- [26] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 165–181, Broomfield, CO, October 2014.
- [27] R. Nigel Horspool and Nenad Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, August 1980.
- [28] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, et al. Crellym: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 631–645, Philadelphia, PA, June 2018.
- [29] Kbuild - The Linux Kernel Documentation. <https://docs.kernel.org/kbuild/kbuild.html>. Accessed on December 13, 2022.
- [30] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and Liveness of MCS Lock—Layer by Layer. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS 2017)*, pages 273–297, November 2017.
- [31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.
- [32] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*, pages 234–248, Cascais, Portugal, January 2019.
- [33] KVM Contributors. Tuning KVM. [https://www.linux-kvm.org/index.php?title=Tuning\\_KVM&oldid=173911](https://www.linux-kvm.org/index.php?title=Tuning_KVM&oldid=173911), June 2018. Accessed on December 13, 2022.
- [34] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, pages 348–370, Dakar, Senegal, April 2010.
- [35] Xavier Leroy. The CompCert C Verified Compiler: Documentation and User’s Manual. <https://compcert.org/man/>, November 2022. Accessed on December 13, 2022.
- [36] Liyi Li and Elsa L Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020)*, pages 7:1–7:29, Dagstuhl, Germany, November 2020.
- [37] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.
- [38] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

- [39] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.
- [40] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 465–484, Carlsbad, CA, July 2022.
- [41] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Manki Yoon. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL 2020)*, January 2020.
- [42] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Automated Verification of Concurrent Code with Sound Semantic Extensibility. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44:1–39, May 2022.
- [43] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, October 2019.
- [44] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 252–269, Shanghai, China, October 2017.
- [45] Raspberry Pi. Kernel Source Tree for Raspberry Pi-provided Kernel Builds. <https://github.com/raspberrypi/linux/tree/rpi-5.4.y>. Accessed on December 13, 2022.
- [46] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of 2020 IEEE Symposium on Security and Privacy (IEEE S&P 2020)*, pages 983–1002, San Francisco, CA, May 2020.
- [47] Jonathan Protzenko, Jean Karim Zinzindhoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F\*. In *Proceedings of the ACM on Programming Languages*, volume 1, pages 1–29, August 2017.
- [48] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 106–113, Vienna, Austria, July 2014.
- [49] Redis Labs. Memtier Benchmark. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark). Accessed on December 13, 2022.
- [50] Helgi Sigurbjarnarson, James Bornholt, Nicolas Christin, and Lorrie Faith Cranor. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI 2016)*, pages 1–16, Savannah, GA, November 2016.
- [51] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.
- [52] SUSE. Performance Implications of Cache Modes. <https://documentation.suse.com/sles/12-SP5/html/SLES-all/cha-cachemodes.html>. Accessed on December 13, 2022.
- [53] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Fred Chong, and Ronghui Gu. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*, pages 641–656, June 2022.
- [54] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

- [55] The Coq development team. The Coq Proof Assistant. <http://coq.inria.fr>. Accessed on December 13, 2022.
- [56] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A Practical Verification Framework for Preemptive OS Kernels. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, pages 59–79, Toronto, ON, Canada, July 2016.
- [57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS 2015)*, San Diego, CA, February 2015.
- [58] Alon Zakai. Emscripten: an LLVM-to-JavaScript Compiler. In *Proceedings of the 26th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications Companion (Wavefront 2011)*, pages 301–312, Portland, Oregon, October 2011.
- [59] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 427–440, New York, NY, January 2012.
- [60] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACl\*: A Verified Modern Cryptographic Library. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, October 2017.
- [61] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 259–274, Huntsville, ON Canada, October 2019.





# Verifying vMVCC, a high-performance transaction library using multi-version concurrency control

Yun-Sheng Chang, Ralf Jung,<sup>†</sup> Upamanyu Sharma,  
Joseph Tassarotti,<sup>∇</sup> M. Frans Kaashoek, and Nickolai Zeldovich

MIT CSAIL    <sup>†</sup> ETH Zurich    <sup>∇</sup> New York University

## Abstract

Multi-version concurrency control (MVCC) is a widely used, sophisticated approach for handling concurrent transactions. vMVCC is the first MVCC-based transaction library that comes with a machine-checked proof of correctness, providing clients with a guarantee that it will correctly handle all transactions despite a complicated design and implementation that might otherwise be error-prone. vMVCC is implemented in Go, stores data in memory, and uses several optimizations, such as RDTSC-based timestamps, to achieve high performance (25–96% the throughput of Silo, a state-of-the-art in-memory database, for YCSB and TPC-C workloads). Formally specifying and verifying vMVCC required adopting advanced proof techniques, such as logical atomicity and prophecy variables, owing to the fact that MVCC transactions can linearize at timestamp generation prior to transaction execution.

## 1 Introduction

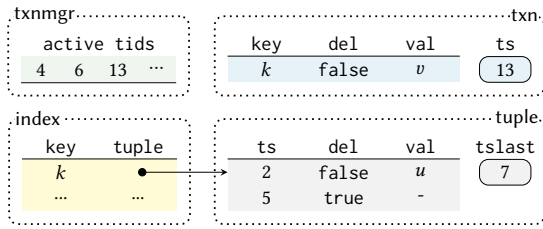
Applications routinely rely on databases not just for storing data durably on disk, but also for ensuring that transactions execute atomically despite concurrency and crashes. This simplifies application development, because the application developer no longer has to worry about concurrency bugs or partial state left over after a crash. Indeed, this pattern is so ubiquitous that it is common for cloud providers to offer databases as a black-box service to application developers. In this model, application correctness and performance crucially hinges on the database system correctly handling all possible corner cases and doing so efficiently.

Achieving both correctness and high performance in a database system for many concurrent transactions is challenging. In particular, when transactions read and write an overlapping set of data items, the database system must ensure the transactions appear to execute in a serial order. A widely used technique for improving performance in this setting is *multi-version concurrency control*, or MVCC [8, 30, 35, 36], in which the database stores not just the latest version of a data item, but also past versions. Storing past versions allows the database system to execute writes that add a new version, while also being able to use the older versions to execute reads from transactions that appear to execute earlier in the serial order.

Multi-version concurrency control requires a sophisticated implementation of its data structures, in order to efficiently track multiple versions of each tuple, implement garbage collection (GC), etc. The implementation must also employ low-level optimizations to get high performance. For instance, using a mutex on a shared counter to get a unique ID for each transaction is too costly, and highly scalable implementations must use contention-free approaches such as relying on the CPU timestamp counter. The end result, therefore, is a complex implementation that can have bugs leading to incorrect or non-serializable executions. These bugs can be costly: they can cause data to be lost or corrupted; they can lead to many applications being affected; and tracking down bugs in the database system can be difficult for application developers.

This paper presents vMVCC, a high-performance MVCC-based transaction library with a formal specification and a machine-checked proof of correctness. vMVCC addresses the core technical challenges faced by the transaction layer in a database, and can be used to build transactional applications. Verifying vMVCC requires addressing several challenges. First, we must formalize a specification that captures the guarantees provided by MVCC transactions in a concise manner. Second, we must develop proof techniques to show that MVCC achieves a serializable execution order in the presence of concurrency. Finally, we must be able to formally reason about high-performance implementations that use low-level programming techniques such as sharded data structures, accessing the CPU timestamp counter with an RDTSC-like instruction, etc.

The key technical challenge addressed in vMVCC lies in dealing with the fact that MVCC’s linearization point happens before the transaction body runs—the linearization point is when the timestamp is obtained in `Begin()`. This makes it challenging to verify MVCC-based transactions because, at the linearization point, the transaction has not executed yet, and the proof does not know what data the transaction is going to write or whether it is going to commit or abort. However, it is important for the specification and proof to update the abstract state of the system at the linearization point, because subsequent transactions must observe these changes. In contrast, under two-phase locking, a transaction linearizes at the point when it commits, where it is well known what state the transaction modified and that it is about to commit.



**Figure 1:** Overview of the main vMVCC data structures. Implementation details are not shown on the figure (e.g., the index and the active transaction IDs are partitioned into multiple shards for scalability).

vMVCC addresses this challenge by adopting *prophecy variables* [1, 18]. Our use of prophecy variables allows the proof to speculatively predict what state the transaction is going to modify and whether it will commit. This translates into the proof considering every possible prediction, allowing vMVCC to update the abstract state accordingly at the linearization point. As the transaction is about to commit, the proof can check whether the prediction was correct or not, and either stop considering further an incorrect prediction, or continue with a correct prediction. vMVCC is not the first to develop or use prophecy variables—many earlier frameworks developed support for them and proved that they are a sound proof technique—but it is the first to prove the correctness of MVCC-based transactions.

We implemented vMVCC in Go, and verified it using the Goose and Perennial frameworks. vMVCC implements sophisticated optimizations such as the use of RDTSC to generate strictly increasing timestamps, on-the-fly GC of past versions, and efficient data structures for storing multiple versions. vMVCC provides a transactional key-value store interface, similar to Silo [35]. For the YCSB benchmark with 32 worker threads, vMVCC achieves an aggregated throughput of 18.6M–52M transactions per second, which is 38–96% of that achieved by the unverified Silo database. For TPC-C, vMVCC achieves a throughput of 10.7K–33K transactions per second per warehouse, which is 25–43% of Silo’s throughput.

The key technical contribution of vMVCC lies in demonstrating how to formally reason about transactions whose linearization point precedes the execution of their transaction body, using prophecy variables. This verification technique would be applicable to any system that uses MVCC [8, 10–12, 14, 19, 27, 30, 32, 35–37]. The second contribution is vMVCC itself, the first verified MVCC transaction library. The vMVCC artifact is interesting in its own right, providing a high-assurance and high-performance implementation, and can be used as a Go package independent of verification. vMVCC includes several other technical contributions, including a verified algorithm for computing strictly increasing transaction IDs using RDTSC, and a precise specification of a transaction library interface using logical atomicity [16].

One of the limitations of vMVCC is that it does not implement durability. In-memory databases are widely used in practice, but we do plan to extend vMVCC to store data durably on disk so that it persists across crashes, and to for-

mally verify it using techniques from Perennial [3]. Another limitation of vMVCC is that it provides a simple key-value data model, as opposed to SQL’s relational data, and does not support range scans.

## 2 Design and interface of vMVCC

vMVCC is a transaction library, and applications interact with it through a standard interface for transactions, as follows (in Go syntax):

```
func (db *DB) Begin() *Txn
func (txn *Txn) Write(key K, value V)
func (txn *Txn) Delete(key K)
func (txn *Txn) Read(key K) (V, bool)
func (txn *Txn) Commit() bool
func (txn *Txn) Abort()
```

vMVCC uses an MVCC design closest to the original protocol as proposed by Reed [30] (also known as multi-version timestamp ordering [36]). The design is based around assigning a strictly increasing timestamp in `Begin()` to every transaction, and storing multiple versions for each key, corresponding to a range of timestamps for which that version is valid. When an application modifies a key, using `Write(k, v)` or `Delete(k)`, the vMVCC transaction keeps track of the modification in a per-transaction write buffer. When an application invokes `Read(k)`, the transaction first checks its local write buffer for pending writes to `k`; if there are no pending writes, it then searches from the global state the version of key `k` whose timestamp immediately precedes the transaction’s timestamp. On successfully calling `Commit()`, the transaction creates a new version for each key in the write buffer with the transaction’s timestamp as well. On calling `Abort()`, or a failed `Commit()`, the transaction drops its write buffer.

Read-only transactions always succeed in vMVCC because vMVCC retains all past versions required by active transactions (i.e., those that have begun but not yet committed or aborted). A transaction involving updates, however, might fail to commit if another transaction with a higher timestamp has read or updated the modified key in the meantime. The reason this requires aborting the first transaction is that, to achieve linearizability, the second transaction should have seen the update made by the first one, but it did not.

**Data structures.** Figure 1 shows the data structures that vMVCC uses to implement its design. The crux of multi-versioning lies in the data structure *tuple*, consisting of a list of versions, a `tslast` field to detect conflicts, and a mutex (not shown) used for synchronizing access to this data structure. Each version corresponds to a range of timestamps for which it is valid, represented by the `ts` field, which marks the start of the validity region. The version is valid until the next version’s `ts` field, or, if this is the last version in the tuple, then it is the latest version. Each version also contains the value (`val`) and whether this key is deleted or not (`del`). The `tslast` field of each tuple represents the highest timestamp of any transaction that has read or written this tuple. It is

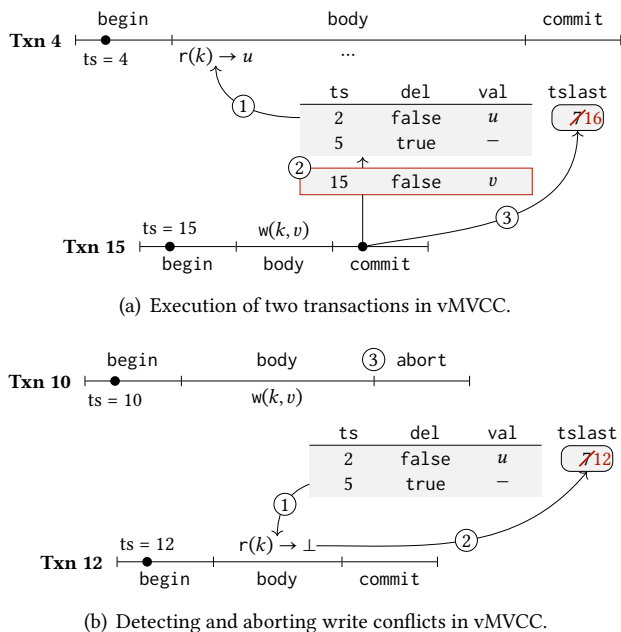


Figure 2: Two example executions of concurrent transactions in vMVCC.

used to detect conflicts if a Write or a Delete with an earlier timestamp tries to commit later on. On top of tuples, vMVCC maintains its *index*, a hash map from keys to tuple pointers. Keys not present in the index are assumed to be not present (deleted) at every timestamp.

Every active transaction in vMVCC is represented using a *transaction object*, which consists of a unique timestamp *ts*, as well as a local write buffer keeping track of the modifications made by this transaction so far. When the transaction commits, it tries to acquire the mutexes of the tuples to be modified, and if successful, atomically applies the modification in its local writer buffer. Transaction IDs are generated by the *transaction manager*. For the purposes of GC, it also keeps track of the IDs of active transactions.

**Execution examples.** Figure 2(a) illustrates an example of two concurrent transactions accessing the same key *k*. The tuple in the example corresponds to key *k*. ① Txn 4 reads the value *u* from the tuple, as the timestamp of the corresponding version (i.e., 2) immediately precedes that of Txn 4. ② Txn 15 writes *v* to *k* by appending a new version tagged with its timestamp at commit time, and ③ increases *tslast* to  $15 + 1$ , preventing transactions with timestamp below 16 from modifying this tuple.

This example also shows the concurrency advantages of MVCC over conventional concurrency control approaches such as two-phase locking (2PL) and optimistic concurrency control (OCC). With 2PL, Txn 15 cannot commit until Txn 4 commits, at which point the lock on *k* is released. With OCC, Txn 4 would have to abort as the value of *k* changes during the execution of Txn 4.

Figure 2(b) shows an example of how vMVCC detects and aborts conflicting writes. ① Txn 12 reads the second

version of the tuple, and ② increases the *tslast* field to its timestamp. ③ Txn 10 attempts to commit and update the tuple, but fails because the timestamp of Txn 10 is less than *tslast* of the tuple (i.e., 12). Thus, Txn 10 aborts.

**Garbage collection.** To reclaim space occupied by unusable versions, vMVCC employs a garbage collector that runs in the background to remove those versions. The garbage collector must ensure that the versions it removes cannot be accessed by any transactions, including those that have not even begun. Concretely, the garbage collector first determines a lower bound on the transaction IDs of all active and future transactions. This lower bound can be computed by finding the minimal transaction ID among the active ones; if there are no active transactions, the current timestamp is used. Because timestamps are strictly increasing (as described below), the garbage collector can safely remove versions whose lifetime ends before that lower bound.

**Generating timestamps with CPU timestamp counter.**

A key requirement for vMVCC is that every transaction is assigned a strictly increasing timestamp. However, assigning these timestamps by modifying a shared in-memory counter leads to contention on that counter. Instead, vMVCC uses the CPU timestamp counter (e.g., RDTSC on x86 machines) to generate timestamps in a scalable way. Modern hardware ensures that timestamps are monotonically increasing and consistent across cores and sockets [2].

One complication is that two threads running on different cores may obtain the same timestamp. vMVCC addresses this problem by using *transaction sites* to make transaction IDs unique. Each site has its own ID, which is a short integer value (e.g., from 0 to 63). When the transaction manager wants to assign a timestamp, it replaces the low bits of the timestamp counter with the site ID value. To ensure that the transaction manager does not use the same site for two transactions at the same time, vMVCC maintains an array of mutexes, one per site, and the transaction manager holds the site's mutex while computing the timestamp. The transaction manager can pick any site ID, such as the one associated with the local core. vMVCC takes a more flexible approach by assigning each thread a site ID in a round-robin manner. Having per-site mutexes ensures that the transaction manager does not contend when assigning timestamps on different sites.

Naïvely replacing the low bits of the timestamp counter with the site ID leads to subtle correctness issues. For example, Txn A may choose the highest possible site ID (all ones), quickly execute, and commit. Txn B, runs after Txn A but chooses the lowest possible site ID (all zeroes). The processor ensures that the RDTSC value seen by Txn B is higher than that seen by Txn A, but once the low bits are replaced with all-ones and all-zeroes, it may be that Txn B's transaction ID is lower than that of Txn A. One possible fix would be to represent the transaction ID as a tuple of the complete



64-bit RDTSC value and the site ID. However, since transaction IDs are used throughout vMVCC, this leads to a noticeable performance overhead.

Instead, vMVCC modifies the timestamp algorithm to ensure that timestamps are strictly increasing. To obtain a timestamp, the transaction manager first obtains  $t$ , the current RDTSC value, and then computes the next highest value  $t' \geq t$  such that  $t'$  has the desired site ID in the low bits. The transaction manager then spins in a loop calling RDTSC until it returns a timestamp  $t'' > t'$ . The transaction manager then uses  $t''$  as the transaction's ID. The reason this loop-based design achieves strictly increasing transaction IDs is that the transaction manager is holding the site's mutex while the CPU timestamp counter passed through  $t'$ . This means no other thread could have generated the same transaction ID. In practice, of course, the loop runs for a few cycles at most, since the RDTSC value will quickly exceed the loop threshold.

**Whole-transaction execution.** For developer convenience, vMVCC provides an interface that wraps up the details of beginning, committing, and aborting a transaction, in `db.Run`, a higher-order function whose implementation is as follows:

```
func (db *DB) Run(body func(txn *Txn) bool) bool {
    t := db.Begin()
    commit := body(t)
    if commit {
        return t.Commit()
    } else {
        t.Abort()
        return false
    }
}
```

The developer provides the body of the transaction, which can use `Read`, `Write`, and `Delete` to access the system state. The transaction body returns a boolean to indicate whether it wants to commit or abort.

### 3 Using and specifying vMVCC

vMVCC is a transaction library that facilitates building and verifying applications by providing an atomic transaction abstraction. We begin with constructing on top of vMVCC an example application that atomically transfers some amount from one account to another, along the lines of what a bank application might do (§3.1). We then describe the formal specification of vMVCC and how to build arbitrary applications on top of it (§3.2).

#### 3.1 Example: AtomicXfer

Figure 3 shows the implementation of `AtomicXfer` (ignore the inline proof for now). This code is implementing a simple bank, transferring `amt` from the `src` account to `dst`. If not enough funds are available in `src`, the transaction aborts. vMVCC ensures that the logical effect of the transaction body, `xfer`, appears to apply atomically. This frees the developer from worrying about other concurrent transactions that

```
// { src ↦ v_s * dst ↦ v_d }
func xfer(txn *Txn, src, dst, amt uint64) bool {
    // { src ↦ v_s * dst ↦ v_d }
    sbal, _ := txn.Read(src)
    // { src ↦ v_s * dst ↦ v_d ∧ sbal = v_s }
    if sbal < amt {
        // { src ↦ v_s * dst ↦ v_d ∧ sbal < amt ∧ ... }
        return false
    }
    // { src ↦ v_s * dst ↦ v_d ∧ sbal = v_s ∧ sbal ≥ amt }
    txn.Write(src, sbal - amt)
    // { src ↦ v_s - amt * dst ↦ v_d ∧ ... }
    dbal, _ := txn.Read(dst)
    // { src ↦ v_s - amt * dst ↦ v_d ∧ dbal = v_d ∧ ... }
    txn.Write(dst, dbal + amt)
    // { src ↦ v_s - amt * dst ↦ v_d + amt ∧ ... }
    return true
}
// If returning false, then { T }
// Else { src ↦ v_s - amt * dst ↦ v_d + amt }

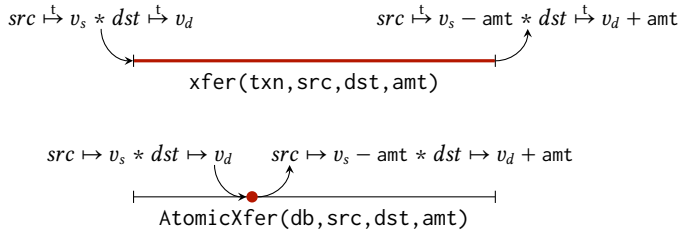
// ⟨ src ↦ v_s * dst ↦ v_d ⟩
func AtomicXfer(db *DB, src, dst, amt uint64) bool {
    body := func(t *Txn) bool {
        return xfer(t, src, dst, amt)
    }
    return db.Run(body)
}
// If returning false, then ⟨ src ↦ v_s * dst ↦ v_d ⟩
// Else ⟨ src ↦ v_s - amt * dst ↦ v_d + amt ⟩
```

Figure 3: Implementation and proof of `AtomicXfer` using vMVCC library.

may affect the balance in `src` or `dst`, or about the versioning going on inside of vMVCC. vMVCC also ensures the transactions execute in a linearizable order, so that once `AtomicXfer` returns, any subsequent transactions will observe the effects of this `AtomicXfer`.

**Sequential reasoning in `xfer`.** vMVCC formalizes the fact that the developer need not consider other concurrent transactions by allowing the developer to use sequential reasoning for the body of the transaction. To achieve this, vMVCC uses Iris [17], a modern concurrent separation logic (CSL) [29], to specify its interface. In Iris/CSL, threads can own logical *resources*, and resource ownership can be exclusive, meaning that if one thread owns a resource, no other thread can own the same resource. For example, the resource  $k \mapsto v$  says that the value of  $k$  is  $v$ , and also says that the current thread *owns*  $k$ —that is, no other thread can own  $k \mapsto v$  in the meantime (and thus no other thread can read or write  $k$ ).

In our example, the proof of the transaction body, `xfer`, assumes ownership of  $src \mapsto v_s * dst \mapsto v_d$ ; the  $*$  operator (“separating conjunction”) says the thread owns both resources and they are disjoint. Having ownership of these resources allows the proof to assume that it is the only one accessing `src` and `dst`. This, in turn, allows the developer to prove `xfer` as if it was running in isolation, with no other concurrent transactions. The overall specification for `xfer` is that, starting with  $\{src \mapsto v_s * dst \mapsto v_d\}$ , if `xfer` runs and terminates, then it either returns false to abort the transaction, or it returns true to commit, and the resources are



**Figure 4:** Figurative specifications of `xfer` and `AtomicXfer`. We highlight the duration of owning the resources with red.

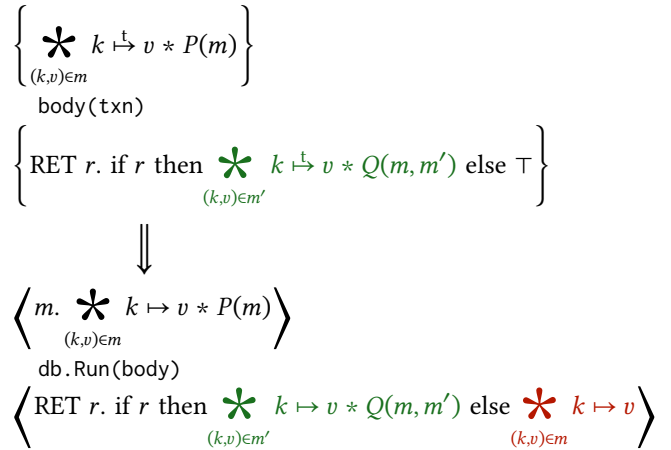
now  $\{src \mapsto v_s - amt * dst \mapsto v_d + amt\}$ . To prove this, the developer considers each line of code, and how that code affects the resources owned by the thread, as shown in the proof state comments between lines of code.

**Concurrent specification for `AtomicXfer`.** Specifying a function with exclusive resource ownership (like we did with `xfer`) simplifies the reasoning for that function, but at the cost of limiting its implementations to sequential ones—only the thread owning the required resources would be allowed to execute the function.

To specify the behavior of `AtomicXfer` in Iris/CSL without requiring ownership of `src` and `dst` for the entire duration of `AtomicXfer`, vMVCC uses the notion of *logical atomicity* [16]. Figure 4 shows the flow of resources in a logically atomic specification of `AtomicXfer` as compared to that of sequential `xfer`. The `xfer` specification says that the thread owns the resources throughout the entire execution of `xfer`, whereas in `AtomicXfer`, the specification says that there will be some point in time at which `AtomicXfer` appears to run atomically. One notable difference here is the kinds of resources appearing in the two specifications. Intuitively, the  $k \mapsto v$  used by `xfer` says that “this transaction believes the value of  $k$  is  $v$ ”, whereas the  $k \mapsto v$  used by `AtomicXfer` reflects “the actual value of  $k$  is  $v$ ”. We will explain the meaning of these resources in more depth in §3.2.

The resulting logically-atomic specification for `AtomicXfer` captures that any number of threads are allowed to concurrently invoke `AtomicXfer`, possibly with overlapping `src` and `dst` values. For each thread’s invocation of `AtomicXfer`, the specification says that the transfer will execute correctly and atomically. The application can, in turn, prove that this maintains some application-level invariant, such as the sum of the balances of all accounts remains fixed.

**Proving `AtomicXfer`.** Proving `AtomicXfer` involves two parts. First, the developer proves that `xfer` meets its specification, as described above. Second, the developer uses the vMVCC library to obtain a proof that `AtomicXfer`’s specification is the logically-atomic equivalent of `xfer`’s sequential specification, as shown in Figure 3. The next subsection describes how vMVCC formally specifies `db.Run` in the general case to enable this second step.



**Figure 5:** Specification of `db.Run`. The angle brackets indicate a logically atomic specification [16]. The vertical arrow indicates that, as a precondition for invoking `db.Run`, the developer must prove the standard Hoare-logic specification shown above the arrow for `body`. Not shown is the part of the specification that describes the representation predicates. We color the resources established for commit with green, and for abort with red.

### 3.2 Specifying the transaction interface

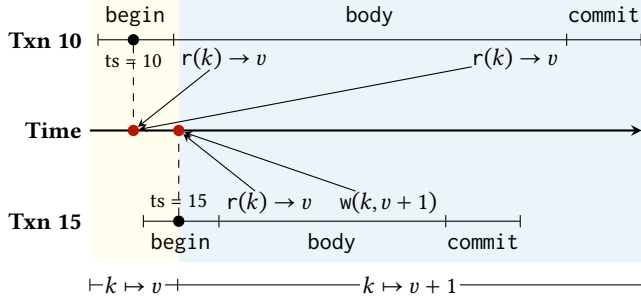
Transactions give users an illusion that they are “isolated” from each other. To capture this intuition, we define the resource  $k \mapsto v$  (which already showed up in the above example) as the *transaction-local view* of the system state. We can then specify operations that manipulate the transaction-local view in terms of  $k \mapsto v$ :

$$\begin{aligned} & \left\{ k \mapsto v \right\} \text{txn.Read}(k) \left\{ RET\ v.\ k \mapsto v \right\} \\ & \left\{ k \mapsto v \right\} \text{txn.Write}(k, u) \left\{ k \mapsto u \right\} \\ & \left\{ k \mapsto v \right\} \text{txn.Delete}(k) \left\{ k \mapsto \perp \right\} \end{aligned}$$

These specifications use standard Hoare-logic syntax, where  $\{P\} op \{Q\}$  means that, if `op` runs starting with the resources specified in precondition  $P$ , it will return with the resources as specified in the postcondition  $Q$ .

Next, we define the resource  $k \mapsto v$  as the *logical view* of the system state, representing the linearizable state. The fact that only a single value of each key is exposed to users might seem counter-intuitive in the case of MVCC, given that the system physically stores multiple values for each key. However, from the application’s point of view, it suffices to view the abstract state of the system as having a single value for each key at any given point in time, and updating that value at the transaction’s linearization point. (We discuss this in more detail in §4.2.)

The specification of `db.Run` shown in Figure 5 connects these two kinds of resources. This is also the top-level theorem of vMVCC as a transaction library. The specification requires the developer to prove a sequential specification for `body` with a precondition that takes the transaction-local view of some set of key-value pairs,  $m$ , along with some



**Figure 6:** An example of two concurrently running vMVCC transactions. Both transactions appear to execute their reads and writes at their linearization points (marked as red). The reason linearization points appear at timestamp generation is that if Txn A linearizes (i.e., runs across its linearization point) before Txn B, then all reads and writes of A should appear to happen before that of B. This is precisely what timestamps are intended to do.

constraints on those values, represented by the predicate  $P(m)$ . The postcondition of body says that, if it chooses to commit, then it should return the transaction-local view of  $m'$  with some constraints  $Q(m, m')$  on how these key-value pairs relate to the starting state.

Given such a specification for body, the specification of `db.Run` says that `db.Run(body)` will be the logically atomic equivalent: at some instant during its execution, it will swap the logical view of  $m$  satisfying  $P(m)$  for that of  $m'$  satisfying  $Q(m, m')$ . Further, if this transaction aborts (either at its own will or because of conflicts with another transaction), then `db.Run(body)` keeps the logical view of  $m$  intact.

As an example, we can instantiate  $P$  and  $Q$  for `AtomicXfer` from §3.1 as follows:

$$P(m) \triangleq \text{dom}(m) = \{\text{src}, \text{dst}\}$$

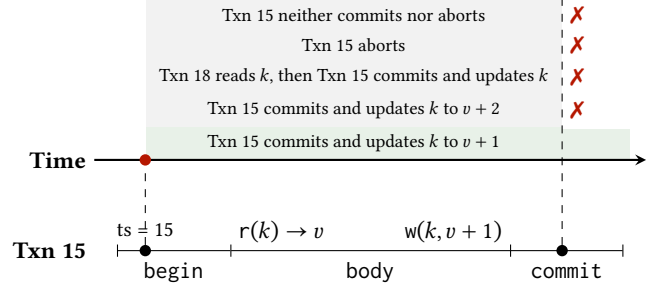
$$Q(m, m') \triangleq m'[\text{src}] = m[\text{src}] - \text{amt} \wedge m'[\text{dst}] = m[\text{dst}] + \text{amt}$$

The use of  $P$  and  $Q$  as arbitrary predicates allows the `db.Run` specification to capture the behavior of body and transfer it to the logically atomic specification of `db.Run(body)`. One technicality here is that  $P$  and  $Q$  are both *pure* predicates, meaning they cannot encode ownership of other resources, but merely restrict the values of  $m$  and  $m'$ .

The specification of `db.Run` can be regarded as a program-logic formalization of strict serializability [15] in the database literature. Serializability comes from the part of the specification that says transactions appear to observe and modify the system state one at a time (at their linearization point), with strictness owing to the fact that they do so during the course of their respective execution (and hence the serial order respects the transaction precedence order).

## 4 Proving vMVCC

This section describes the important aspects of our proof for vMVCC. We start with a key verification challenge and how we solve it with prophecy variables (§4.1). We describe how



**Figure 7:** Transaction futures, showing several example futures speculated through prophecy variables and their interaction with prophecy resolution.

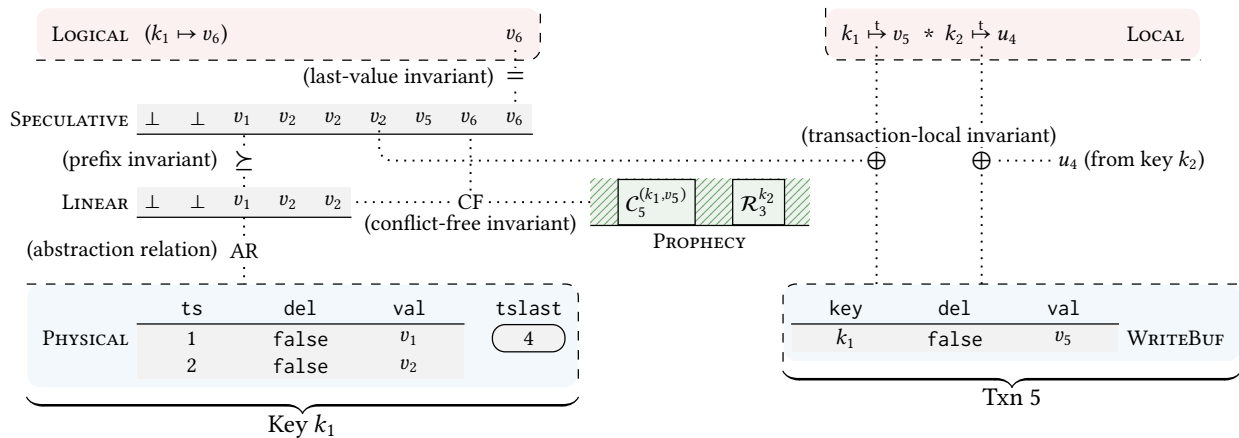
we abstract a tuple from its physical representation containing multiple versions to its logical view with a single value, which potentially reflects some update that happens only in the future (§4.2). We present a key invariant about the prophecy variable used in vMVCC, and how the invariant helps maintain other system-wide invariants under correct and incorrect predictions (§4.3). We discuss how we define the transaction-local view of the system state, and its connection to the logical view (§4.4). We finally conclude this section with the challenges and the approach regarding proving strict monotonicity of transaction IDs (§4.5).

### 4.1 Speculation using prophecy variables

We introduce the verification challenge with an example shown in Figure 6. Observe that in the example, the value of  $k$  to be read by Txn 10 is determined up front by  $k \mapsto v$  at its linearization point, despite the fact that by the second read of Txn 10, Txn 15 has already committed and updated the physical state of  $k$ . Similarly, the write of Txn 15 updates the logical state to  $k \mapsto v + 1$  before it physically executes. This kind of “speculative” behavior of MVCC turns out to be tricky to reason about in a Hoare-logic reasoning style where the proof considers each line of code in turn and reasons about how that code updates the abstract and physical states.

The challenge arises from the fact that MVCC transactions linearize when their timestamp is generated. In the proof, the logical state must be updated at the transaction’s linearization point, which happens before the transaction body runs. The changes to the logical state depend both on the transaction itself (i.e., what data the transaction decides to write), as well as conflicts with other transactions (i.e., whether another transaction reads or writes the same keys as this transaction in a way that will force this transaction to abort, as discussed in §2). This poses the question: how do we know, at the transaction’s linearization point, what values will a transaction write, and whether a transaction will encounter a conflict and thus be forced to abort? To tackle this issue, we use prophecy variables.

Intuitively, prophecy variables allow the proof to speculate about future execution. In the case of vMVCC, the prophecy variable is a list of transaction actions, which describes what actions each transaction will perform, and in what order.



**Figure 8:** Overview of vMVCC’s key physical states (in blue regions), logical states (in red regions), intermediate states, and the system-wide invariants relating them (the dotted lines). We introduce the tuple abstraction relation, the prefix invariant and the last-value invariant in §4.2, the conflict-free invariant in §4.3, and the transaction-local invariant in §4.4. Here  $u_4$  is the value of the speculative view of  $k_2$  at timestamp 5.

We refer to the list as the *future-action list*. There are two kinds of actions in vMVCC’s proof:  $C_t^m$  (“Txn  $t$  commits and applies updates  $m$  to the system state”) and  $R_t^k$  (“Txn  $t$  reads key  $k$ ”). Transaction aborts are represented by a commit with an empty write-set.

At transaction begin time, vMVCC’s proof uses the prophecy variable to speculatively predict the execution of the transaction, which allows the proof to update the logical state as if it knew what the transaction is going to do. The main challenge of using the prophecy variable, however, is that some of the predictions could be *incorrect*—it predicts something that does not match what happens later. As an example, Figure 7 shows five concrete predictions for Txn 15 that increases the value of  $k$  by 1. Only the bottom prediction turns out to be correct when the transaction actually commits. The incorrect predictions eventually diverge from the actual changes made by the transaction, and will make the logical state inconsistent with the physical state.

To deal with the divergence, the proof performs *prophecy resolution* at the point where the transaction actually commits and updates the physical state. Prophecy resolution allows the proof to stop considering cases corresponding to predictions that did not match reality, and continue only with the cases that did. We will elaborate more on correct/incorrect predictions and prophecy resolution with a concrete example in §4.3.

This description may make it sound like there are a large number of cases to consider in the proof, greatly increasing the proof burden. In practice, the predictions are symbolic, rather than concrete timestamps, keys, and values; for instance, the prophecy variable speculates the updates made by a transaction as a symbolic partial map. Furthermore, the proof can group together many speculative executions (e.g., those in which the transaction of interest is speculated to commit without encountering a conflict), and consider the entire family of executions just once.

## 4.2 Incorporating speculation in abstract state

vMVCC exposes a single linearizable copy of the system state, thereby freeing the users from explicitly reasoning about the timestamps. Thus, the logical view (shown in the “logical” row of Figure 8) of vMVCC is a single value for every key, and the proof must connect this logical view to the physical state (shown in the “physical” row of Figure 8), consisting of the Go struct representing each tuple.

This connection is challenging for several reasons, including the fact that the Go data structure contains multiple versions, and the fact that the value in the logical view may not even be present in the Go data structure, if it is made by a write speculated by the prophecy variable for an active transaction. Moreover, reasoning about the physical layout of the tuple in all intermediate proofs is cumbersome.

To address these challenges, we introduce two intermediate layers modeled with *monotonic lists* (i.e., lists that only grow). The first is the *linear view* of the tuple, shown in the “linear” row of Figure 8. The linear view is a contiguous list of values, indexed by timestamps. The linear view gives us an elegant way to specify operations on tuples: reading a tuple with a given timestamp  $t$  just returns its value at index  $t$ . If the transaction needs to extend `tslast`, doing so extends the linear view up to the new `tslast` timestamp, filling in new entries with the last value in the list. Writing a tuple with a given timestamp  $t$  extends the tuple up to index  $t$ , and appends the new value to the end.

To capture the speculative behavior of MVCC as described in §4.1, we add the “speculative” layer, as shown in Figure 8 as well. The *speculative view* is yet another contiguous timestamp-indexed list, much like the linear view, but includes the writes from transactions that have linearized but have not yet finished executing and updating the physical state. The proof looks up and extends the speculative view at the linearization point (the ability for such extension is guaranteed by strict monotonicity of vMVCC’s timestamps),





The second challenge stems from the fact that an inconsistent prediction, such as the example above, involves multiple transactions, and therefore relies on prophecy resolution in multiple threads. The prediction about each individual transaction and thread could be correct in its own right, but it is the combination of them that leads to a contradiction. How should the proof be structured to establish the contradiction despite only doing one prophecy resolution at a time? vMVCC's proof addresses this challenge by maintaining a sufficiently strong invariant that carries along facts from each prediction to derive contradictions against later predictions as needed, as we describe below.

To illustrate this point, we first sketch out the proof for a correct prediction, where a transaction commits and there are no conflicts that would have forced it to abort, and then show how vMVCC's proof handles incorrect predictions.

**Predicted commit without conflicts.** In Figure 9(a), Txn 5 is speculated to commit without encountering any conflict. The reason is that at its linearization point, the commit action of Txn 5 in the future-action list,  $C_5^{(k,v_5)}$  (there might be multiple of them in the list, but the proof cares only about the first one), is *conflict-free* against all the actions prior to it. We define  $C_t^m$  to be conflict-free against an action  $a$  if (1)  $a = \mathcal{R}_{t'}^k$ , where  $t' \leq t \vee k \notin \text{dom}(m)$ , or (2)  $a = C_{t'}^{m'}$ , where  $t' < t \vee \text{dom}(m') \cap \text{dom}(m) = \emptyset$ . Knowing that Txn 5 will commit without conflicts, the proof safely extends the speculative view up to timestamp 5 using the old value  $v_2$ , and appends the new value  $v_5$  to it (which updates the logical view to  $v_5$  as well) without violating the *conflict-free invariant*, as described below.

Intuitively, the conflict-free invariant requires that a transaction reflects its update to the speculative view only if the first commit action of the transaction is conflict-free against all the actions prior to it in the future-action list. As we will see below, this invariant is crucial to prove invariance of the prefix property between the linear and speculative views.

On reading key  $k$ , the proof resolves the head of the future-action list to  $\mathcal{R}_5^k$ . Then, it uses the conflict-free invariant to deduce that transactions which contain updates to the speculative view, but not to the linear view, must have timestamps greater than or equal to the timestamp of this read. This implies that the speculative view can differ from the linear view only after a timestamp  $t > 5$ , allowing the proof to re-establish the prefix invariant after extending the linear view. A similar reasoning goes for commit, except the proof additionally uses the promise that Txn 5 will commit to know the value at timestamp 5 of the speculative view is  $v_5$ .

**Predicted commit despite conflicts.** In Figure 9(b), Txn 5 is speculated to commit despite the presence of conflicts because its first commit action,  $C_5^{(k,v_5)}$ , conflicts with an earlier action  $C_6^{(k,v_6)}$ . The proof, as in the previous case, extends the speculative view up to timestamp 5 using the old value  $v_2$ ; however, it does not append the new value  $v_5$  as doing so

would violate the conflict-free invariant, and proceeds as if the transaction will abort, which makes the invariant true.

For read, the proof of the prefix property is similar to the previous case. For commit, however, the proof cannot re-establish the prefix property after extending the linear view, because it indeed did not apply the new value  $v_5$  at the linearization point. Fortunately, at this point the proof knows two facts that contradict each other: (1) reaching the prophecy resolution point for commit, the execution must have passed the conflict detection as illustrated in Figure 2(b), implying the length of the linear view  $l \leq 5 + 1$  (the +1 part is due to our lists being zero-indexed); (2) some conflicting action (in this case  $C_6^{(k,v_6)}$ ), which extends the linear view to at least timestamp  $t > 5$ , must have happened *before* Txn 5 commits, implying  $l > 5 + 1$ . The proof closes this case with the derived contradiction.

#### 4.4 Abstract state of a transaction

As mentioned in §4.1 (and illustrated in Figure 6), the value of key  $k$  to be read by a transaction is determined up front by  $k \mapsto v$  at the transaction's linearization point. Reading from the physical state, however, happens only at some later point in time, and the value is based on  $k \mapsto v'$ , as specified in §3.2. This means the proof has to somehow connect  $k \mapsto v$ ,  $k \mapsto v'$ , and  $v''$ , the result of physically reading the tuple of  $k$ . This section describes how the system-wide invariants shown in Figure 8 establish that connection.

Let us first consider the case where the transaction has not written key  $k$ . Our first step then is to show  $v = v'$ . Recall that at the linearization point of Txn  $t$  that reads or writes  $k$ , we extend the speculative view of  $k$  up to  $t$  using its last value. Doing so, along with the last-value invariant, allows us to deduce that the value of the speculative view at index  $t$  is  $v$  (and will remain so since the speculative view is monotonic). The proof then follows from the definition of the *transaction-local invariant*, which says that if Txn  $t$  has not written  $k$ , then  $v'$ , the transaction-local value, is equal to the value of the speculative view at index  $t$ .

Our next step is to show  $v' = v''$ . Again recall that physically reading the tuple of  $k$  at timestamp  $t$  means extending the linear view of  $k$  up to  $t$  (if the value at  $t$  is still absent), and looking up its value at index  $t$ . The proof of  $v' = v''$  then follows immediately from the prefix invariant that requires the linear view to remain a prefix of the speculative one.

Now consider the case where the transaction has last written  $k$  with value  $u$ . As specified in §3.2, the logical effect of the write is  $k \mapsto u$ . We thus define the remaining case for the transaction-local invariant: if the transaction has written  $k$ , then the transaction-local value is equal to the value in its local write buffer.

The contents of the write buffer are also what the speculated updates in a commit action (i.e.,  $m$  in  $C_t^m$ ) resolve to. This allows us to obtain the equality between the speculated updates with the actual updates at the prophecy resolution

point, which is crucial when re-establishing the prefix invariant as discussed in §4.3.

#### 4.5 Strict monotonicity of transaction ID

Another challenge in vMVCC’s proof is establishing strict monotonicity of transaction IDs, for vMVCC’s RDTSC-based algorithm described in §2. The challenge lies not only in proving that the algorithm generates strictly increasing transaction IDs, but also in being able to logically execute the transaction (i.e., update the logical state) at the linearization point corresponding to that transaction ID. The reason this is challenging is that the linearization point for some transaction ID  $t'$  might not correspond to any line of code that was executed for this transaction—the algorithm simply spins in a loop waiting for RDTSC to advance past  $t'$ , and linearization occurs when *any* transaction observes that  $t'$  has passed.

To formally reason about this algorithm, vMVCC’s proof maintains a logical table of slots, one per timestamp. The slot contains the logical set of changes that a transaction with that timestamp wants to perform, represented as a ghost function. The actual state changes performed by this ghost function are determined by prophecy variables, as described above. The proof uses the slots to invoke the ghost functions for each timestamp in order, as the RDTSC clock advances; the proof maintains a “latest-slot” timestamp corresponding to the last table slot that has been invoked. The invariant associated with this proof states that this latest timestamp is always below (or equal to) the real RDTSC clock. Furthermore each future slot is protected by the site’s mutex that corresponds to this timestamp.

When the transaction manager first computes  $t'$ , it registers the  $t'$  slot in the table, putting in a ghost function that will perform its transaction’s changes. Since  $t' > t$ , the proof has not yet invoked the ghost function for this slot, and the current thread also holds the site’s mutex needed to fill this slot (which proves that no concurrent thread could fill the same slot). As the transaction manager runs the loop waiting for RDTSC to move past  $t'$ , it updates the latest-slot with each iteration, executing all of the ghost functions in the slots that have been advanced over. The proof takes advantage of *later credits* [31] in Iris that enable verification of this “unsolicited helping” pattern.

The invariant for the slot table says that, for every slot with a timestamp below the latest-slot, its ghost function callback has been invoked. As a result, when the transaction manager’s loop exits, it knows that the latest-slot is at least as high as  $t'$ , and therefore its ghost callback must have been invoked (either by this same thread or by some other thread running the same loop).

## 5 Implementation and proof details

We implemented vMVCC in Go, and verified its implementation using the Perennial framework [3] (based on Iris [21–23] and Coq [33]), using Goose [4] to lift vMVCC’s Go code into

Component	Lines of code (Go) / proof (Coq)
Tuple	260 / 1947
Transaction	419 / 4489
Index	85 / 496
Timestamp	24 / 311
Misc	39 / 361
Ghost state	- / 947
Global invariants	- / 2566
Total	827 / 11117

Figure 10: Lines of code and proof for each component of vMVCC.

Perennial. To enable vMVCC’s proofs of MVCC transaction linearizability, we incorporated prophecy variable support from Iris [18] into Perennial.

Figure 10 summarizes the implementation and proof effort, not including changes to Perennial that were necessary for the verification. The lines of proof include the specifications for each function in vMVCC’s implementation. The proof effort for vMVCC required about 13× as many lines of proof as lines of code, which is in the same ballpark as other verified systems that handle concurrency [3, 6, 13].

The implementation contains several low-level optimizations that improve performance. We used RDTSC to generate transaction IDs. We also padded data structures to avoid false cache-line sharing that limits multi-core scalability, and sharded the index and the set of active transaction IDs to reduce contention from concurrent accesses.

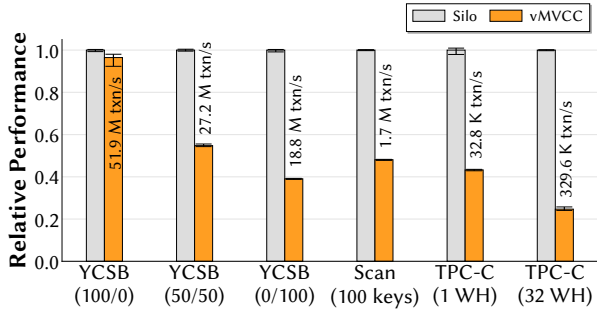
### 5.1 Bugs found during verification

When we were first designing and implementing vMVCC, we were careful to structure the code in a way that makes it clear why the code is correct, what the invariants are, how they are maintained, and what guarantees each interface or function provides. Nevertheless, during the actual verification, we ran into several bugs in corner cases that we missed or did not correctly handle in the implementation, highlighting the importance of formal reasoning. In this subsection, we give several examples of such bugs.

One interesting bug we found when verifying vMVCC is related to garbage collection. The buggy code is:

```
func (site *TxnSite) getSafeTS() uint64 {
    site.mutex.Lock()
    var tidmin uint64 = MAX_U64 /* buggy */
    // var tidmin uint64 = site.getCurrentTS()
    for _, tid := range site.tidsActive {
        if tid < tidmin {
            tidmin = tid
        }
    }
    site.mutex.Unlock()
    return tidmin
}
```

When the garbage collector starts a new round of GC, it first calls `getSafeTS` on each site to collect the per-site minimal transaction ID, and then computes a globally minimal one



**Figure 11:** Comparison of Silo and vMVCC. For YCSB, each transaction reads or writes a key sampled from a uniform distribution with a certain R/W ratio. For TPC-C, the number of warehouses is same as the number of worker threads.

among them. If every transaction site is empty (i.e., if every site returned `MAX_U64`), the garbage collector generates a timestamp using an arbitrary site. (Recall that vMVCC always places a site ID in the low bits of the timestamp, and the choice of site ID does not matter, as it is purely there to ensure uniqueness.) The bug arises when a transaction enters the system right after `getSafeTS` returns, and then the garbage collector computes a timestamp larger than the ID of that transaction. Our fix to this bug is to generate a timestamp within each site, as shown in the commented-out code. Doing so ensures that future transaction IDs generated by this site will be larger than the one `getSafeTS` returns.

Another subtle bug we discovered is missing the wait loop when generating transaction IDs, violating the strict monotonicity of our timestamp generation scheme. The fix was the looping `RDTSC` algorithm described in §2. Finally, since our protocol is centered around timestamps, we also discovered several off-by-one errors in the implementation when conducting the verification of vMVCC (e.g., where greater-than comparisons should have been greater-than-or-equal-to comparisons).

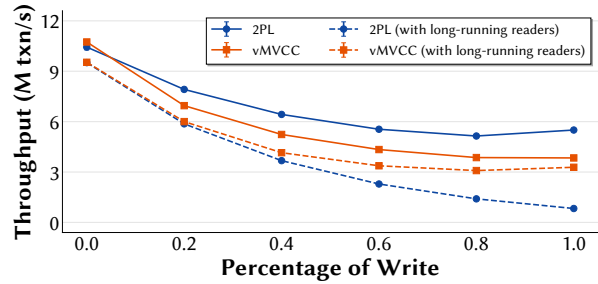
## 6 Evaluation

We experimentally answer the following questions:

- Is vMVCC competitive with state-of-the-art unverified systems? (§6.2)
- Does the use of MVCC in vMVCC help with long-running read-only transactions? (§6.3)
- Are the low-level optimizations in vMVCC important for performance? (§6.4)
- Does vMVCC scale under high-contention workloads? (§6.5)

### 6.1 Experimental setup

All experiments were done on an AWS EC2 `c5.9xlarge` instance with 36 vCPUs (18 physical CPUs, each shared by 2 hardware threads via hyper-threading) and 72 GB of main memory, running Linux 5.15.0 and Go 1.20.3.



**Figure 12:** Comparison of 2PL and vMVCC under YCSB (4 keys accessed per transaction,  $\theta = 0.85$ , 24 threads), with and without 8 long-running reader threads that repeatedly read 1% of the entire key space.

We used the YCSB benchmark [7] to understand the performance characteristics of vMVCC under various workloads. Unless otherwise specified, we execute each YCSB put or get in a separate transaction. The data set contains 1M key-value pairs with each key being an 8-byte integer and value an 100-byte string. The access pattern follows the uniform distribution, or the Zipfian distribution, with a parameter  $\theta$  controlling the skewness of the distribution. We vary the read-write ratio and the number of keys accessed in each transaction.

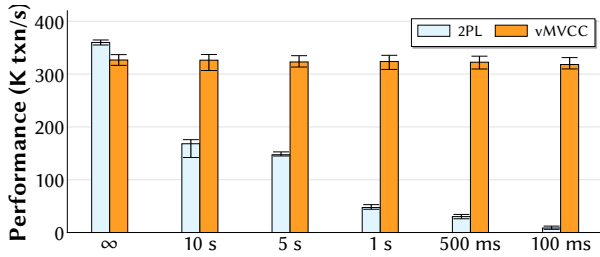
We also used the TPC-C benchmark, which involves more sophisticated transactions. TPC-C models the operation of a wholesale supplier, a common online-transaction processing (OLTP) workload. It contains 9 tables and 5 kinds of transactions, each with various workload characteristics. In particular, most transactions can be processed in a single warehouse, so it is natural and efficient to map each warehouse to one thread. Our current implementation of vMVCC requires the key to be an 8-byte integer, and every tuple needs a key. Because of these limitations we made two modifications to TPC-C. First, we do not support “get customers by their last name” appearing in the `OrderStatus` and `NewOrder` transactions; they are replaced with just “get customers by customer ID”. Second, the `History` table does not have a primary key, so we randomly generate one for it.

We employ a background GC thread for vMVCC in every experiment. We repeat each experiment 10 times, each for 30 seconds. We report the mean, minimum, and maximum (the last two as error bars) among the 10 runs.

### 6.2 Comparison with Silo

To evaluate whether vMVCC achieves competitive performance with state-of-the-art systems, we compare vMVCC to Silo [35], a high-performance transactional database system. Because vMVCC does not store data durably, we compare with MemSilo, a variant of Silo that does not persist its data. Silo is an OCC/MVCC based system, using OCC to provide serializability, and using MVCC to access a consistent snapshot of old versions. Unlike vMVCC, Silo does not generate a new version for every write, but only once per *snapshot epoch* (on the order of 1 second), which reduces memory management costs. Silo’s OCC/MVCC design has the ad-





**Figure 13:** Comparison of 2PL and vMVCC under TPC-C (32 warehouses), with a thread periodically invoking the read-only transaction StockScan.

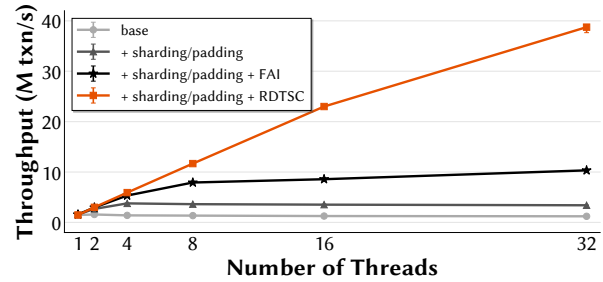
vantage of lower memory usage and allocation overhead over vMVCC’s pure MVCC design. On the other hand, Silo only ensures its snapshot transactions (those that access past versions) always read a consistent snapshot, without imposing ordering constraints on them with respect to normal linearizable transactions, whereas in vMVCC, a “snapshot transaction” is simply a linearizable transaction that does not perform writes.

Figure 11 shows the results of the comparison for several configurations of YCSB and TPC-C, normalizing to the throughput achieved by Silo. Similarly to Silo, each worker thread in vMVCC generates the workload parameters and then immediately processes the transaction. We used a YCSB profile where each transaction accesses a single key sampled from a uniform distribution. vMVCC achieves 96.6% the throughput of Silo for a read-only workload in YCSB, and 38.8% for a write-only workload. For TPC-C, vMVCC achieves 43% the throughput of Silo for 1 warehouse and 25.7% for 32 warehouses. We hypothesize that the performance difference between Silo and vMVCC is largely due to (1) vMVCC’s higher memory allocation overhead for storing past versions, and (2) its inefficient way of executing range scans—lacking a tree-like index structure, vMVCC relies on the continuity invariant of TPC-C [34], and expands a range query into multiple point queries. To test these hypotheses, we conducted the following two experiments.

First, we ran the same write-only YCSB workload, except that we fixed the write value to some statically allocated string, and modified vMVCC to perform in-place update on its tuples, without changing any other parts of the code (hence the resulting system is not even correct, but it is merely for us to understand more about vMVCC’s performance characteristics). Applying these changes increases the relative performance from 38.8% to 87.3%.

Second, we ran an additional range scan workload where each transaction first randomly picks a starting key, and then reads the next 100 keys. Silo executes each transaction with a single scan, whereas vMVCC issues 100 reads. Figure 11 shows the results in the “scan” column. vMVCC achieves 48.1% of Silo’s throughput; the difference mostly attributes to more cache misses in vMVCC.

Based on these experiments, we conclude that the gap between vMVCC’s performance and that of Silo is indeed



**Figure 14:** Throughput of vMVCC with different optimizations enabled. The benchmark is YCSB (1 key read per transaction,  $\theta = 0.2$ ).

largely due to memory allocation and vMVCC’s lack of support for range scans. For benchmarks that do not stress these two aspects, vMVCC achieves performance competitive with Silo.

### 6.3 Robustness to long-running readers

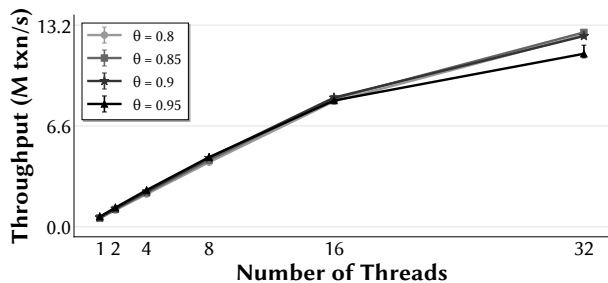
One main advantage of MVCC over traditional concurrency control protocols is that its performance should remain stable even in the presence of long-running readers. To confirm that vMVCC’s design indeed achieves these performance benefits, we implemented a variant of vMVCC that uses two-phase locking for concurrency control instead of MVCC, and compared the performance of vMVCC with this 2PL variant.

**YCSB.** We first compare vMVCC and 2PL under the YCSB workload, using a YCSB profile where each transaction reads or writes 4 keys. We fixed the number of threads to 24,  $\theta$  to 0.85, and varied the read-write ratio from 0% to 100%. We then ran one experiment without long-running readers, and another one where the workload includes 8 transactions repeatedly reading 10K keys (1% of the entire key space).

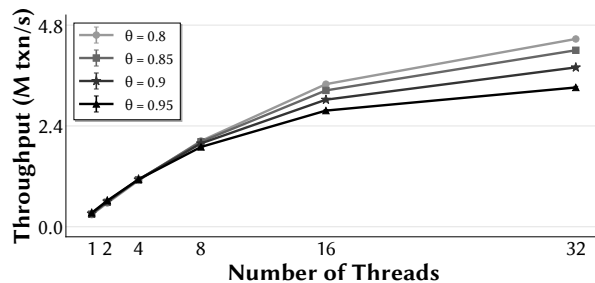
Figure 12 shows the results. In the absence of long-running readers, 2PL performs better than vMVCC for all read-write ratios except for the read-only workload (comparing the solid lines). The difference stems from MVCC’s overhead of (1) generating timestamps and (2) keeping past versions and the associated memory allocation costs.

In the presence of long-running readers (comparing the solid and dashed lines of each system), vMVCC’s throughput drops slightly between the range of 11.5%–22.2%, whereas 2PL’s throughput drops significantly as the write ratio increases (e.g., 72.6% and 84.9% for write ratio 80% and 100%, respectively). As a result, the performance of 2PL with long-running readers is worse than that of vMVCC for workloads with 40% or more writes; for instance, under write ratios 80% and 100%, vMVCC performs 2.2 $\times$  and 4 $\times$  better than 2PL, respectively. The reason is that, in 2PL, the long-running readers hold read locks on keys for a long duration, preventing other transactions from writing to those keys.

**TPC-C.** We also compare vMVCC and 2PL under the TPC-C workload. Similarly to prior work [36], we add a read-only transaction StockScan that counts the number of each item in



(a) Read scalability of vMVCC under high-contention workloads.



(b) Write scalability of vMVCC under high-contention workloads.

Figure 15: Scalability analysis under high contention. The benchmark is YCSB (4 keys accessed per transaction).

all warehouses. We parametrize the workload by the interval of invoking StockScan. Figure 13 shows the results.

When no StockScan is invoked (represented by the  $\infty$  interval on the x-axis), 2PL performs better than vMVCC by around 14%. However, when there are StockScan transactions running, NewOrder transactions that update the stock table will conflict with StockScan, and block under 2PL concurrency control. As the interval between StockScan transactions decreases, 2PL’s performance drops significantly, whereas vMVCC throughput remains more-or-less the same, since StockScan accesses old versions of tuples and does not impact other transactions. For StockScan intervals 500 and 100 ms, the throughput of vMVCC is 11 $\times$  and 54.4 $\times$  that of 2PL. In terms of latency, vMVCC maintains its 99.9% latency around 3.4 ms across all StockScan intervals, whereas the 99.9% latency of 2PL increases from 3.2 ms in the absence of StockScan transactions, to a few tens and occasionally hundreds of ms when StockScan is invoked every 100 ms.

## 6.4 Low-level optimizations

vMVCC implements (and verifies) two low-level optimizations to achieve high performance on many cores: (1) padding and sharding data structures and mutexes, to avoid cache-line contention, and (2) using RDTSC to generate transaction IDs without shared-memory contention. To understand whether they are important for performance, we enable each optimization in turn and measure the resulting performance. To stress the implementation, we chose a lightweight YCSB profile where each transaction accesses a single key. We chose a low-contention setting ( $\theta = 0.2$ ) so that transactions largely access different portions of the key space; we will evaluate scalability under high contention in the next subsection.

To evaluate the benefit of the RDTSC-based transaction ID generation, we compare with two alternatives. The first is a lock-based design where the transaction manager acquires a mutex to get (and increment) the next transaction ID counter. The second is a lock-free implementation that uses the fetch-and-increment (FAI) instruction to atomically obtain the next transaction ID.

Figure 14 shows the results. The optimizations have little effect on a single core, but significantly improve vMVCC’s performance on 32 cores. Partitioning and padding index and transaction sites improves vMVCC’s performance by 2.8 $\times$  at 32 cores. Using FAI increases throughput by a further 3 $\times$  over the lock-based design at 32 cores. Finally, RDTSC-based transaction IDs achieve yet another 3.7 $\times$  improvement in throughput compared to FAI at 32 cores. In summary, the results show that all of these optimizations are important for scaling vMVCC’s performance with many cores.

Enabling all the optimizations, vMVCC’s throughput scales by 15.6 $\times$  using 16 threads. The result suggests that vMVCC eliminates almost all contention on its internal data structures (when the keys themselves do not contend). The throughput scales further by 1.66 $\times$  when doubling the number of threads to 32, implying that vMVCC can benefit from hyper-threading even though not as much as from having more physical cores.

## 6.5 Scalability under contention

The previous section showed that vMVCC scales nearly linearly for a low-contention workload, with its low-level optimizations. In this section, we evaluate vMVCC’s scalability under high-contention workloads, using a YCSB profile where each transaction issues 4 reads/writes, with the skewness parameter  $\theta$  ranging from 0.8 to 0.95.

Figure 15 shows the results. For reads (Figure 15(a)), before reaching the hyper-threading threshold (i.e., 18 cores), the throughput scales almost linearly with respect to the number of threads, except for extremely contended workloads (e.g.,  $\theta = 0.95$ ): the aggregated throughput of 16 threads is 14.9 $\times$  that of a single thread for  $\theta = 0.8$ , and 12.6 $\times$  for  $\theta = 0.95$ . Scalability drops after reaching the hyper-threading threshold because of interference, especially for higher skewness: using 32 threads achieves 22.8 $\times$  better performance for  $\theta = 0.8$ , and 16.9 $\times$  for  $\theta = 0.95$ .

For writes (Figure 15(b)), besides from hyper-threading interference, having more contention also causes more conflicts between transactions, and hence higher abort rates. For instance, the abort rate at 32 threads for  $\theta = 0.8$  is 4.8%, whereas for  $\theta = 0.95$  is 27.6%. The result is that vMVCC’s

throughput with 32 threads is 11.7× that of a single thread for  $\theta = 0.8$  and 9.9× for  $\theta = 0.95$ .

The results show that vMVCC’s performance scales with the number of cores even for workloads of high contention.

## 7 Related work

vMVCC is the first formally verified MVCC-based system, but builds on prior work on formal verification and specification of transactions, as we now discuss.

**Verified systems.** The closest related work to vMVCC is GoTxn [6], a verified transaction library that uses 2PL for concurrency control. GoTxn stores data durably to disk and uses the verified GoJournal [5] journaling system to provide crash atomicity. vMVCC uses a more sophisticated concurrency control plan (MVCC), which allows it to achieve high performance for long-running read-only transactions, while GoTxn uses standard 2PL which does not perform well with long-running readers. vMVCC also implements and verifies sophisticated optimizations, such as strictly increasing RDTSC-based timestamps, which are not present in GoTxn.

Malecha et al. [28] verified a simple relational database, focusing on SQL queries, the relational data model, and the use of B+-trees on disk. These issues are complementary to the focus of vMVCC, which targets handling concurrent transactions using sophisticated concurrency control protocols and low-level optimizations.

**Prophecy variables.** Abadi and Lamport [1] first proposed prophecy variables as a proof technique to establish refinement mappings between state machines. Jung et al. [18] later integrated it in a Hoare-style program logic. Prior work using prophecy variables is mostly focused on verification of protocols and small examples of data structures and algorithms, such as RDCSS, the Herlihy-Wing queue [18], and the atomic snapshot algorithm [24].

In this paper, we apply prophecy variables in a sophisticated transaction library. We use prophecy variables to make more elaborate predictions about the behavior of transactions, including what data they read and write, and we demonstrate that prophecy variables are useful for reasoning about transactions.

### Framework for specifying and verifying transactions.

Lesani et al. [25] develop a framework for verifying software transactional memory systems and apply it to the NOrec transactional memory algorithm [9]. NOrec uses a form of OCC, in which transactions check whether they have been invalidated by conflicting writes during commit time. As with 2PL, NOrec transaction’s linearization point occurs during commit, and hence does not appear to require prophecy variables in its proof.

vMVCC uses logically atomic triples to specify transactions, instead of classical serializability and linearizability definitions [15] that are based on trace equivalence (e.g., as

used by GoTxn). This makes it easier to verify clients of a transaction library by proving Hoare triples about code running inside of the transaction library. Prior work has similarly found it useful to introduce alternate specifications for transactions and serializability in the context of formal verification. The Push/Pull model [20] provides a set of primitive operations which can be used to describe a variety of transactions. Any system that can be decomposed into these operations is guaranteed to be serializable. C4 [26] is a framework that supports verifying transactional objects, that is, concurrent data structures that allow chaining multiple operations together in an atomic transaction. The framework supports composing transactional objects as components of a higher-level transactional object.

## 8 Conclusion

This paper presented vMVCC, the first MVCC-based transaction library with a machine-checked proof of correctness. A key challenge in verifying vMVCC lies in reasoning about the linearization of transactions under MVCC, where the linearization point occurs before the transaction body actually runs. vMVCC addressed this challenge by using prophecy variables to speculate whether upcoming transactions are going to commit, and what values they are going to write, thereby allowing vMVCC to state and prove a simple yet general specification for its top-level transaction interface using logical atomicity. vMVCC incorporates further low-level optimizations, such as using RDTSC to generate strictly increasing transaction IDs, with corresponding proofs of correctness, to achieve high performance. An evaluation demonstrated that, for a range of YCSB and TPC-C workloads, vMVCC’s throughput is 25–96% of the throughput of Silo, a state-of-the-art unverified system; that vMVCC benefits from MVCC to achieve good performance for long-running read-only transactions compared to two-phase locking; and that vMVCC’s low-level optimizations are important for achieving high performance. At the same time, vMVCC’s proof effort—13× as many lines of proof as lines of code—is on par with other verified concurrent systems.

## Acknowledgments

We are grateful to Anish Athalye, Sanjit Bhat, Alexandra Henzinger, Jon Howell, Derek Leung, the anonymous reviewers, and our shepherd, Adriana Szekeres, for their valuable feedback that improved this paper. We thank Tej Chajed for discussions on transactions and the Perennial framework. This work was supported by a grant from Amazon AWS through the Science Hub program, and by NSF awards CCF-2123864 and CCF-2318722. The code and proof of vMVCC is available at:

<https://pdos.csail.mit.edu/projects/vmcc.html>

## References

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual IEEE Symposium on Logic in Computer Science*, pages 165–175, Edinburgh, Scotland, July 1988.
- [2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Sept. 2014.
- [3] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, Oct. 2019.
- [4] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, Jan. 2020.
- [5] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, Virtual conference, July 2021.
- [6] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 447–463, Carlsbad, CA, July 2022.
- [7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, June 2010.
- [8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang. Spanner: Google’s globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.
- [9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 67–78, Bangalore, India, Jan. 2010.
- [10] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server’s memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, NY, June 2013.
- [11] etcd Authors. etcd API, Apr. 2023. <https://etcd.io/docs/v3.6/learning/api/#revisions>.
- [12] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Kohala Coast, HI, Aug.–Sept. 2015.
- [13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, Nov. 2016.
- [14] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, May–June 2015.
- [15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.
- [16] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, Austin, TX, Jan. 2011.
- [17] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.
- [18] R. Jung, R. Lepigre, G. Parthasarathy, M. Rapoport, A. Timany, D. Dreyer, and B. Jacobs. The future is ours: prophecy variables in separation logic. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, pages 45:1–45:32, New Orleans, LA, Jan. 2020.
- [19] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, June–July 2016.



- [20] E. Koskinen and M. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 186–195, Portland, OR, June 2015.
- [21] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming (ESOP)*, pages 696–723, Uppsala, Sweden, Apr. 2017.
- [22] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, Jan. 2017.
- [23] R. Krebbers, J. Jourdan, R. Jung, J. Tassarotti, J. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 77:1–30, St. Louis, MO, Sept. 2018.
- [24] L. Lamport and S. Merz. Prophecy made simple. *ACM Transactions on Programming Languages and Systems*, 44(2):6:1–6:27, Apr. 2022.
- [25] M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR)*, page 516–530, Newcastle upon Tyne, UK, Sept. 2012.
- [26] M. Lesani, L. Xia, A. Kaseorg, C. J. Bell, A. Chlipala, B. C. Pierce, and S. Zdancewic. C4: verified transactional objects. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–31, 2022.
- [27] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, May 2017.
- [28] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, Madrid, Spain, Jan. 2011.
- [29] P. W. O’Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.
- [30] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Sept. 1978. <http://hdl.handle.net/1721.1/16279>.
- [31] S. Spies, L. Gäher, J. Tassarotti, R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Later credits: Resourceful reasoning for the later modality. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Ljubljana, Slovenia, Sept. 2022.
- [32] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, Portland, OR, June 2020.
- [33] The Coq Development Team. *The Coq Proof Assistant, version 8.15*, Jan. 2022. URL <https://doi.org/10.5281/zenodo.5846982>.
- [34] Transaction Processing Performance Council (TPC). TPC benchmark C standard specification, revision 5.11, Feb. 2010. [https://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-c\\_v5.11.0.pdf](https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf).
- [35] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.
- [36] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, Mar. 2017.
- [37] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppapa, X. Su, and V. Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pages 2653–2666, Virtual conference, June 2021.

# Automated Verification of Idempotence for Stateful Serverless Applications

Haoran Ding<sup>1</sup>, Zhaoguo Wang<sup>1</sup>, Zhuohao Shen<sup>1</sup>, Rong Chen<sup>1,2</sup>, and Haibo Chen<sup>1</sup>

<sup>1</sup>Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

<sup>2</sup>Shanghai AI Laboratory

## Abstract

Serverless computing has become a popular cloud computing paradigm. By default, when a serverless function fails, the serverless platform re-executes the function to tolerate the failure. However, such a retry-based approach requires functions to be idempotent, which means that functions should expose the same behavior regardless of retries. This requirement is challenging for developers, especially when functions are stateful. Failures may cause functions to repeatedly read and update shared states, potentially corrupting data consistency.

This paper presents Flux, the first toolkit that automatically verifies the idempotence of serverless applications. It proposes a new correctness definition, *idempotence consistency*, which stipulates that a serverless function's retry is transparent to users. To verify idempotence consistency, Flux defines a novel property, *idempotence simulation*, which decomposes the proof for a concurrent serverless application into the reasoning of individual functions. Furthermore, Flux extends existing verification techniques to realize automated reasoning, enabling Flux to identify idempotence-violating operations and fix them with existing log-based methods.

We demonstrate the efficacy of Flux with 27 representative serverless applications. Flux has successfully identified previously unknown issues in 12 applications. Developers have confirmed 8 issues. Compared to state-of-the-art systems (namely Beldi and Boki) that log every operation, Flux achieves up to 6× lower latency and 10× higher peak throughput, as it logs only the identified idempotence-violating ones.

## 1 Introduction

A serverless application typically comprises a collection of functions, which may be stateful. For example, they may communicate with each other through a shared database. Major serverless platforms generally support the stateful model, such as AWS [17], Microsoft [59], and Google [36]. Platforms generally employ a retry-based fault tolerance mechanism for stateful applications — they automatically retry a function in case of an unexpected error [18, 35, 57].

However, this mechanism mandates developers to write

*idempotent* applications that produce consistent results irrespective of the number of retries. In a sequential system that invokes functions sequentially, developers can reason about each function independently. However, a concurrent system can invoke functions simultaneously. Therefore, developers must consider all possible interleavings of concurrent functions, making it challenging to write idempotent applications.

This paper presents Flux, the first toolkit that automatically verifies the idempotence of concurrent serverless applications. Building such a toolkit posed several challenges. First, a formal idempotence definition for concurrent systems is desired but currently missing. Second, automated verification requires examining all possible interleavings of concurrent serverless functions with arbitrary failures, which is prohibitively expensive. Third, for non-idempotent applications, the toolkit should accurately identify the code that corrupts idempotence, enabling developers to fix the issues.

To overcome the first challenge, we propose a novel idempotence definition for concurrent systems — *idempotence consistency*. A serverless application is idempotence-consistent if, for any observable behavior of an execution with retries, there exists another execution without retries that can produce the same behavior. Achieving idempotence consistency makes clients unaware of retries during execution (Section 3). Unlike alternative idempotence conditions, such as exactly-once execution [45, 73], idempotence consistency offers greater flexibility. An idempotence-consistent application does not necessarily ensure exactly-once execution of *all* database operations.

To tackle the second challenge, we propose *idempotence simulation* to realize compositional proof, which enables proving the idempotence consistency of an application by verifying each function individually. For each function, Flux verifies that every possible execution with retries has a corresponding retry-free execution that can simulate it (Section 4). Existing work [66] can realize automated verification in a sequential system by comparing the execution results when retries happen with the results without retries. However, in a concurrent system, verification requires modeling the con-

current environment to account for the side effects of running multiple functions concurrently. Unfortunately, existing modeling approaches are not fully automated as they ask developers for hints, such as invariants [40, 60] and rely conditions [47, 53]. In contrast, Flux automates the generation of rely conditions and other hints. Additionally, Flux proposes *failure reduction* to avoid enumerating all failure cases.

To help fix idempotence issues, Flux can identify idempotence-violating operations whose re-execution corrupts idempotence consistency. Developers can use logs to ensure exactly-once execution semantics for these operations rather than all operations via existing mechanisms [45, 73].

We evaluate Flux on 27 representative serverless applications with 79 functions. These applications are from various sources, such as the AWS serverless application repository [3], a GitHub repository (10.9k stars) [9], popular serverless benchmarks [8, 10], and applications commonly used in papers about serverless computing [5, 6]. Flux successfully identifies previously unknown idempotence issues in 12 applications. Compared to state-of-the-art systems (namely Beldi [73] and Boki [45]) that log all operations, Flux achieves up to 6× lower latency and 10× higher peak throughput, as it logs only identified idempotence-violating operations.

Nevertheless, Flux still has several limitations. First, although we design it for stateful serverless functions, its assumption that states only include data in NoSQL databases restricts its applicability to storage systems of other types. We need to model the semantics of other storage systems carefully. Second, our verification method is sound but incomplete since Flux cannot handle certain serverless functions, such as functions having certain types of unbounded loops. Last, Flux currently supports only Java applications since we build Flux based on a symbolic execution engine for Java applications [61]. Despite these limitations, we believe that Flux takes an important step towards enabling verification for idempotence consistency of serverless functions.

## 2 Motivation and Our Approach

**Why Need Idempotence?** The concept of idempotence is crucial for applications that rely on retry-based methods to tolerate failures. Without idempotence, re-executing failed computations may result in unexpected side effects, causing severe correctness issues [42, 48, 64, 67]. With the emergence of serverless computing, idempotence has become a significant requirement for serverless applications. However, this requirement poses a challenge when serverless functions run concurrently and are stateful, placing a substantial burden on using serverless platforms [44, 69, 73].

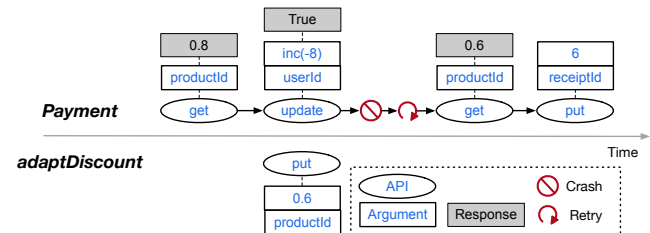
We use an example in Figure 1 to illustrate why re-executing a non-idempotent application can cause issues. This is an example derived from a real-world e-commerce web application, Spree [68]. The *payment* function atomically checks the customer’s balance and deducts the price with a discount rate using a conditional update API (line 4). This

```

1 void payment(productId, userId, price) {
2   discount := get("Discount", productId);
3   total := price * discount;
4   success := cond_update("Balance", userId,
5                       inc(-total), gte(total));
6   receiptId := generateId(userId, productId,
7                       localtime());
8   if(success)
9     put("Receipt", receiptId, total);
10 }
11
12 void adaptDiscount(productId, percent) {
13   if(!isValid(percent))
14     return;
15   discount := 1.0 - percent/100.0;
16   put("Discount", productId, discount);
17 }

```

**Figure 1:** A simplified e-commerce serverless application with two functions. When *balance* in the database is greater than or equal to *total* (*gte(total)* is true), *cond\_update* (line 4) decreases *balance* by *total* (*inc(-total)*) and returns true. Otherwise, it returns false. The *generateId* function (line 6) returns a receipt identifier. The *isValid* function (line 13) returns true iff *percent* is between 0 and 100.

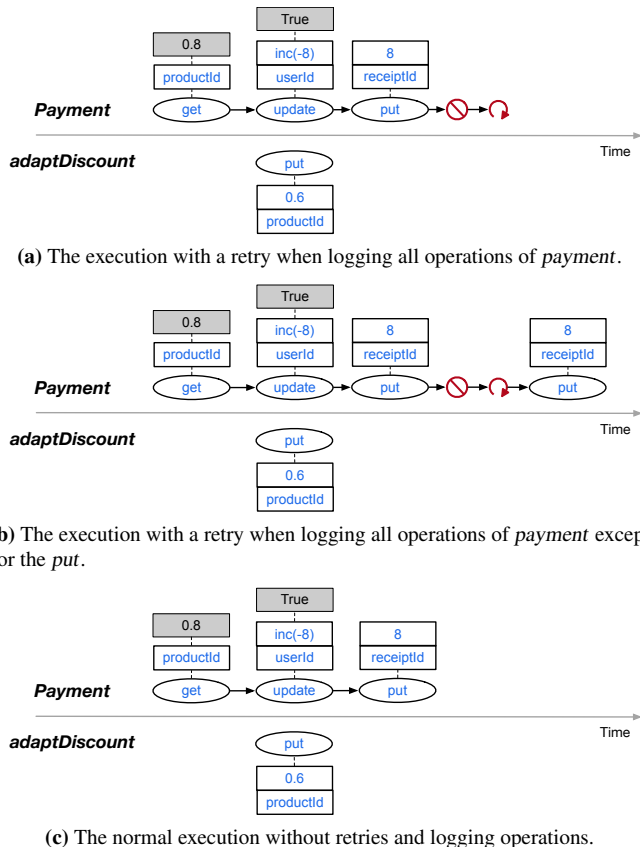


**Figure 2:** The concurrent execution of *payment* and *adaptDiscount* in Figure 1.

method needs to create a receipt accordingly (line 9). Meanwhile, the *adaptDiscount* function changes the discount for a specific product, which the seller typically invokes.

Suppose a failure occurs after *payment* deducts the price from the customer’s balance at line 4 but before it creates a new receipt at line 9. Consider the sequential execution of *payment* without concurrency. If the platform re-executes this function after the failure, the function will deduct the price twice from the customer’s balance. One possible solution to ensure idempotence is to log the conditional update operation. Additionally, it is necessary to log the execution of *generateId*, as *localTime* returns different values on retry. These logs can ensure that the function will not re-execute *update* (line 4) and *generateId* (line 6) on retry, which will always return the same value as the first execution. The solution is enough to provide idempotence under the sequential scenario.

However, the above solution does not work when *payment* runs concurrently with *adaptDiscount*. For example, suppose *adaptDiscount* changes the discount after *payment* fails at line 4 but before its re-execution. Then, *payment* would have deducted the price with the old discount but created a new receipt with the new discount, which poses an inconsistency between the customer’s balance and the corresponding receipt. Figure 2 shows the specific interleaving. Flux aims to automatically find the correct logging strategy under both



**Figure 3:** Three different concurrent executions of the functions in Figure 1. The legend is the same as that in Figure 2.

sequential and concurrent scenarios via verification.

**Idempotence Condition.** Some recent efforts have focused on the retry-based fault tolerance mechanism for serverless applications [44, 45, 69, 73]. Although they optimize runtimes or libraries of serverless computing, they overlook the definition of idempotence. For example, Beldi [73] and Boki [45], which contribute novel distributed logging mechanisms, equate idempotence with executing each database operation exactly once. They achieve this by logging every operation to ensure that functions execute each operation only once. However, repeating some operations does not compromise idempotence. For instance, as shown in Figure 3b, logging all operations except for *put* on *receipt* can still ensure idempotence, as *payment* will always write the same value into the receipt on retry as it did in its first execution. However, as illustrated in Figure 3a, Beldi and Boki need to log every operation, which is over-restricted and incurs unnecessary performance costs. This logging strategy misses the opportunity to maximize performance while ensuring idempotence.

When defining idempotence for serverless applications, we should consider what kind of execution with retries is acceptable. The intuitive requirement is that clients should be unaware of retries. This requirement enables us to define acceptable execution with retries in terms of normal execution

without retries. For instance, the executions with a retry in Figure 3b and Figure 3a are both acceptable because they are equivalent to the normal execution in Figure 3c. However, the execution in Figure 2 is unacceptable because we cannot find an equivalent normal execution for it. The normal concurrent execution of *payment* with *adaptDiscount* will never deduct “8” from the balance but record “6” in the receipt. Therefore, determining whether an application is idempotent requires checking whether any possible execution with retries is acceptable.

**Verification of Idempotence.** Several frameworks for verifying storage systems also prove the idempotence of recovery functions [23, 24, 27, 28, 66]. Specifically, the resulting state of the recovery should be consistent even if the system fails during recovery and retries the recovery function many times. The work based on Crash Hoare Logic [23, 24, 27, 28] verifies idempotence by proving that the crash condition of the recovery function always implies its precondition. Developers need to specify both pre and crash conditions manually. The push-button verification approach [66] frees developers from such a proof burden by automatically verifying recovery functions with SMT solvers. However, all such methods assume that the recovery procedure is sequential. Verifying the idempotence of concurrent functions is still missing.

To prove the idempotence of concurrent functions, we use compositional proof techniques. The fundamental idea is to break down the verification of an application’s idempotence into verifying each function individually. However, the main challenge is defining the property that needs verification for each function, which can facilitate compositional proof. Besides, modeling the behavior of other concurrent function instances also poses a challenge. Existing methods typically use invariants [60] or rely conditions [47, 53] to model the concurrent environment. Invariants describe the properties of the system state that persist during concurrent execution, while rely conditions depict how other concurrent functions can change the system state. Unfortunately, developers must explicitly specify all of them. We need to infer invariants or rely conditions automatically for automated verification.

**Our Approach.** To define idempotence, we propose a new consistency model called *idempotence consistency* (Section 3). An execution with retries is acceptable if there exists another normal execution without retries that can exhibit the same observable behavior (e.g., Figure 3b). An application is idempotence-consistent if all possible executions with retries are acceptable. To verify idempotence consistency, we propose idempotence simulation (Section 4), which extends traditional forward simulation [55] and enables compositional proof for idempotence consistency. Specifically, the verification process tries to find a mapping from each step during the execution with retries to  $n$  steps during the execution without retries such that the single step and the  $n$  steps exhibit the same observable behavior.



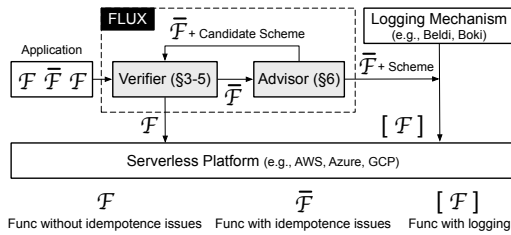


Figure 4: The architecture of Flux.

```

1 bool retry := random();
2 if(retry)
3 {
4   reset_local_state();
5   goto BEGIN;
6 }

```

Figure 5: The pseudocode simulating random failures and retries of a function  $f$ . BEGIN is a label at the beginning of  $f$ .

Figure 4 shows the components of Flux. First, developers provide the source code of a serverless application for Flux. Then, Flux checks each function individually to reason about idempotence simulation (Section 5). If all functions pass verification, the application is idempotence-consistent. If not, advisor identifies operations that corrupt idempotence consistency based on the results of the verifier (Section 6). Developers can use existing logging mechanisms to ensure exactly-once semantics of such operations. Compared with logging all operations pessimistically, Flux guarantees idempotence while reducing unnecessary protection overhead.

### 3 Idempotence Consistency

Idempotence consistency requires that each execution with retries should have the same observable behavior as another execution without retries. To formally define idempotence consistency, Flux uses an automaton to model the concurrent execution of functions. An automaton includes system states and a set of steps. Each step  $(S, e, S')$  represents a state transition from state  $S$  to  $S'$ , which triggers an event  $e$  observable to clients (e.g., a function invocation). Note that some steps may not produce events because they are not observable to clients.

**System State.** The system state consists of the shared state and the local state of each function instance. In scenarios where functions use a NoSQL database, the shared state  $D$  constitutes a collection of key-value pairs stored persistently in databases. Given an instance executing a function, its local state includes the invocation arguments, the local variables, the return value, and the next program statement to execute. The start system state only contains the shared state because the platform has not invoked any functions at the beginning.

**Event.** An automaton may produce an event during each state transition, which is observable to clients. In the context of serverless functions, Flux considers three types of events: function invocation events, function response events, and third-party service events. When creating a new instance  $f_{id}(args)$  to run the serverless function  $f$ , the automaton pro-

duces an invocation event  $(f_{id}, inv(args))$ . When the instance  $f_{id}$  finishes its execution and successfully responds to clients with the value  $v$ , it produces a response event  $(f_{id}, resp(), v)$ . When the serverless function requests a third-party service  $s$ , it produces a third-party service event  $(id_s, args, ret)$ . The  $args$  and  $ret$  represent parameters and return values, respectively. An automaton produces such events when a service has side effects, which developers must explicitly specify.

**Client-Observable Behavior.** The client-observable behavior of an execution includes all events generated during the execution and the final shared state observable by clients. The events include function invocation, response, and third-party service events. We use  $\langle H, D \rangle$  to denote the client-observable behavior, where  $H$  represents the event sequence generated by the automaton throughout the execution, and  $D$  is the shared state reached after the execution.

Given a function set  $F$ , to model the execution of functions in  $F$  under failure, we rely on the following failure model and star operator.

**Failure Model.** Flux assumes that failure can occur at any time during the execution of an instance. The failure of individual instances does not affect the persistent shared state or individual local states of other instances. Furthermore, when the platform retries an instance, it retains the same identifier and arguments, generating no invocation events.

**Star Operator.** Given a function  $f$ ,  $f^*$  denotes a function synthesized by inserting a code fragment after every statement. When random failures and retries occur during a function's execution, the platform will re-initialize the local state and re-execute the function from the first statement (without retry events). The code fragment (Figure 5) simulates random failures by resetting the local state and simulates retries by jumping to the beginning of  $f$ .  $F^*$  is a function set synthesized by applying the star operator to each function in  $F$ .

Based on the above concepts, Flux is able to define idempotence consistency as the relationship between two automata. It means that  $F$  allows all possible client-observable behaviors for the automaton of  $F^*$ .

**Definition 1 (Idempotence Consistency)** For any start system state  $S$  and any step sequence of the automaton executing  $F^*$  from  $S$ , if the step sequence results in the client-observable behavior  $\langle H, D \rangle$ , then there always exists another step sequence of the automaton executing  $F$  from  $S$  such that it also results in  $\langle H, D \rangle$ .

### 4 Proof Strategy

Using compositional proof techniques [52–54, 76], Flux automatically verifies the idempotence consistency of serverless applications. The fundamental idea is to simplify the proof of a concurrent program by reasoning about each of its components separately. Several existing approaches utilize compositional proof to verify the correctness of concurrent programs, such as RGSim [53], AtomFS [76], and Armada [54].

However, the primary difference between Flux and existing approaches is that existing approaches require human experts to aid the proof, such as manually specifying the correctness definition or modeling program behavior under concurrency. Flux, on the other hand, performs entirely automated verification without human intervention.

**Preliminary.** Flux adopts compositional proof techniques, verifying a concurrent program by checking each component individually rather than enumerating all possible interleavings. For instance, when verifying a concurrent stack implementation [52], programmers only need to separately consider the correctness of *push* and *pop* functions under concurrency. To perform compositional proof, programmers first need to manually specify each component’s expected behavior (e.g., specifications for *push* and *pop* functions). Second, programmers should carefully craft the pre- and post-conditions of each statement, which are propositions describing the system state before and after executing the statement. The verification goal is that when the start system state satisfies the pre-condition of the first statement, the system state after executing the component must satisfy the post-condition of the last statement under concurrency. Note that other concurrent threads can simultaneously modify the system state. Therefore, to consider all possible execution results, pre- and post-conditions must cover all possible system states before and after executing a statement. Developers need to prove that the pre- and post-conditions are stable under concurrency, which means they always hold irrespective of how other concurrent functions simultaneously modify the system state. Finally, to verify the stability of pre- and post-conditions, programmers must manually define a rely condition  $R$ , which describes the state transition made by other concurrent threads.  $R$  is a relation of system states. Each  $(S, S')$  in  $R$  indicates that other concurrent threads might change the current state  $S$  to  $S'$ . In the example of the stack, the rely condition specifies the impact of concurrent *push* and *pop* operations on the global linked list that represents the stack. We can define it as  $\{(\ell, \ell') \mid ((\exists v. \ell' = PUSH(\ell, v)) \vee (\ell' = POP(\ell)))\}$ .  $\ell$  represents the state of the linked list, while  $\ell'$  is the new state after applying the operations of  $PUSH(\ell, v)$  or  $POP(\ell)$ .

## 4.1 Idempotence Simulation

Instead of manually crafting the specification, Flux introduces a new correctness definition — *idempotence simulation*. A serverless function  $f$  satisfies idempotence simulation if there exists a forward simulation [55] between  $f^*$  and  $f$  under the same rely condition  $R$ . The forward simulation means that from the same start shared state with the same invocation argument, each step of executing  $f^*$  has zero or multiple corresponding steps of executing  $f$  that can simulate it. If a step  $s$  of  $f^*$  changes the shared state  $D$  to  $D'$  and produces an event  $e$ , a step sequence  $s_1 \dots s_i$  of  $f$  can simulate it if and only if carrying out these steps sequentially from  $D$  also reaches  $D'$  and produces  $e$ . Note that idempotence simulation

does not only consider the shared state reached by executing the current instance but also  $D$  reached by executing other concurrent instances according to  $R$ . Flux differs from previous verification frameworks [23, 24, 27, 28, 66] that focus on the idempotence of sequential functions by considering intermediate states. This difference is significant because, in a concurrent setting, these intermediate states may be externally observable.

Given a function set  $F$ , Flux decomposes the proof of idempotence consistency into reasoning about the idempotence simulation of every function  $f$  in  $F$  based on the following theorem.  $f^* \sqsubseteq_R f$  means that for any start shared state and invocation argument, there exists a forward simulation between  $f^*$  and  $f$  under the same rely condition  $R$ .

**Theorem 1** Given a function set  $F$ , if each function  $f$  in  $F$  satisfies idempotence simulation, then  $F$  satisfies idempotence consistency, denoted as the predicate  $idem(F)$ .

$$(\forall f \in F. \exists R. f^* \sqsubseteq_R f) \rightarrow idem(F).$$

Appendix A presents the formal proof of Theorem 1. We only illustrate its intuition as follows. Existing work [53] based on rely conditions has proposed methods to prove that the specification can exhibit all possible observable behaviors of the implementation. When proving idempotence consistency, we observe that we can treat  $F^*$  as the implementation and  $F$  as its specification. Then we can utilize the compositional proof technique in the existing work to verify idempotence consistency. An important fact is that the steps of implementation consist of the steps from each component, while the specification of the implementation consists of the specification of each component. Therefore, the existing work first proves that for each component and its specification, each step of the component has zero or multiple steps of the specification that can simulate it under the rely condition  $R$ . Then the verifier can compose the proof for each component to imply that each step of the implementation always exhibits the observable behavior allowed by its specification. In the scenario of idempotence consistency, we treat each serverless function  $f^*$  as the component. Then we can treat  $f$  as the specification for  $f^*$ . Based on the observation, Flux defines idempotence simulation as the forward simulation between  $f^*$  and  $f$ , which can imply idempotence consistency.

## 4.2 Automated Concurrency Reasoning

To verify the program with compositional proof, existing approaches [52–54, 76] require programmers to manually define the pre- and post-condition of each statement, as well as the rely condition for concurrent state transitions. The cause of the need for manual effort is that different definitions of correctness usually require different pre-, post-, and rely conditions. Programmers need to deeply understand the definition of correctness to find and specify the appropriate conditions.

However, idempotence consistency presents a unified correctness definition for stateful serverless functions, establish-

ing an opportunity to generate pre-, post- and rely conditions automatically. These conditions capture the potential concurrent accesses to the shared database state and describe the state transition on each access. Flux accomplishes this with symbolic execution. To reduce the complexity of analysis, Flux models the semantics of each API as a sequence of atomic read operations or atomic write operations on a set of key-value pairs, as most serverless platforms use NoSQL databases with key-value interfaces [16, 34, 58]. Then, it can symbolically execute all functions and check the parameters of issued database operations to analyze the data in the shared state that functions can access. Section 5 will depict more details.

Flux identifies three types of rely conditions.

- **Read-only:** all concurrent accesses to a specific key-value pair are read operations. Flux formalizes this type of rely condition as  $(D[k] = v, D[k] = v)$ , which means before and after the access, the value ( $v$ ) indexed by  $k$  in the database keeps unchanged.
- **Arbitrary update:** functions could update the data in the database to arbitrary values. Flux formalizes this type of rely condition as  $(\exists v_1. D[k] = v_1, \exists v_2. D[k] = v_2)$ .
- **Constant update:** functions will update the specific key-value pair to only a constant value. Flux formalizes this type of rely condition as  $(\exists v_1. D[k] = v_1, D[k] = c)$ , where  $c$  is the constant value.

Flux constructs pre- and post-conditions in the Floyd-Hoare style (“ $\{P\}C\{Q\}$ ”).  $C$  is the next program statement to execute.  $P$  is the pre-condition before executing  $C$ , while  $Q$  is the post-condition after the execution. If  $C$  is the first statement of the function, then  $P$  is *true*. Flux adopts the following rules to automatically generate the pre- and post-conditions according to the semantics of  $C$  and different rely conditions:

- $\{P\}put(k,c)\{P \wedge (D[k]=c)\}$ : if the rely condition specifies that all concurrent updates are constant updates with a constant value of  $c$ , then  $D[k]$  will always be  $c$  after executing  $put(k,c)$ .
- $\{P\}v := get(k)\{P \wedge (D[k]=v)\}$ : if the rely condition specifies that all concurrent accesses to  $k$  are read operations, then the value of  $D[k]$  should be exactly  $v$  which is the return value of the  $get$  operation in  $C$ .
- $\{P\}if(P_1(v))\{P \wedge P_1(D[k])\}$ : suppose  $C$  is a branch statement based on  $P_1(v)$ , and  $v$  is the value read from the database, indexed by  $k$ . The post-condition is  $P \wedge P_1(D[k])$  if  $D[k]$  satisfies one of the following requirements according to rely condition: 1)  $D[k]$  is read-only; 2) functions can update  $D[k]$  to a constant value  $c$  such that  $P_1(c)$  is true.
- $\{P\}C\{P\}$ : in the other cases, the post-condition is the same as the pre-condition, which is stable.

### 4.3 Unbounded Loop

Another challenge in automated verification involves dealing with functions that contain unbounded loops. A loop is unbounded when its maximum number of iterations is not constant. Existing approaches require that programmers manually specify loop invariants to handle unbounded loops. However, previous works have shown that finding a proper loop invariant is challenging [19, 33]. Flux reasons about unbounded loops without requiring loop invariants. The following paragraphs provide the details in two cases:

Case 1. The operations in the unbounded loop do not modify the shared state in the database. For this case, Flux treats the entire unbounded loop as an uninterpreted function [12], which is a symbolic function and may return arbitrary values. Specifically, Flux derives a new function  $g$  from the original function  $f$  by replacing the unbounded loop with an uninterpreted function. Then, Flux directly reasons the idempotence simulation for  $g$  instead of  $f$ .

Case 2. The operations in the unbounded loop may modify the shared state. Flux addresses this type of unbounded loop with Theorem 2, which requires that the parameters of the write operations in such unbounded loops must remain the same between normal execution and retry. For convenience, Flux represents a function with an unbounded loop as  $\{C_1; L; C_2\}$ , where  $L$  is the unbounded loop,  $C_1$  denotes all code before the loop, and  $C_2$  denotes all code after the loop.  $B_L$  denotes the loop body of  $L$ . We present the theorem as follows. Appendix C provides the formal proof and an example of applying the theorem.

**Theorem 2** Given a function  $f$  with the unbounded loop in case 2,  $f$  satisfies idempotence simulation if the number of iterations of the loop  $L$  remains unchanged on retry, and  $C_1$ ,  $C_2$  and  $B_L$  can satisfy the following requirements: 1) They all satisfy idempotence simulation; 2) Their inputs do not change on retry; 3) They will not affect the shared state on retry once the function has successfully executed them.

### 4.4 Failure Reduction

The platform may re-execute a function an arbitrary number of times. It is impossible to verify idempotence simulation if we enumerate all possible failure cases, which yields infinite possible executions. Instead, we prove that it is sufficient to verify a function only by examining the executions satisfying two conditions, as stated in the following theorem.

**Theorem 3** For any function set  $F$ , if each function  $f \in F$  satisfies idempotence simulation under the following two conditions: 1) failure happens only after statements modifying the shared state, and 2) failure occurs at most once, then each function  $f \in F$  also satisfies idempotence simulation under arbitrary failure and retries.

This result (i.e., failure reduction) mitigates the challenge of proving idempotence simulation with infinite failure. It

```

1 bool retry := random();
2 if(retry && hasretried < LIMIT)
3 {
4   reset_local_state();
5   hasretried++;
6   goto BEGIN;
7 }

```

**Figure 6:** The pseudocode simulating random failures and retries of  $f^n$ .

transforms the problem of examining executions with infinite failure into the problem of reasoning about executions with finite failure. Moreover, it maintains the soundness of the verification, signifying that Flux does not overlook any idempotence issues.

The intuition behind the first condition is that a statement that does not modify the shared state lacks side effects if a failure occurs after it. Because the failure effectively renders the result of the statement invisible to clients and the following code. Therefore, when the failure occurs, it appears as if the function instance never executed the statement.

The second condition is correct and ensures soundness. We formalize its correctness based on the following concepts. Given a function  $f$ ,  $f^n$  denotes the function whose number of re-execution is not more than  $n$  times ( $n \geq 0$ ). We can construct  $f^n$  by inserting the code fragment in Figure 6 after every statement of  $f$ . The global variable *hasretried* is initially zero, which indicates the number of retries that have occurred. We can simulate  $n$  retries for  $f^n$  by setting the constant *LIMIT* to  $n$ . The correctness of the second condition follows from the following theorem. Appendix B presents the formal proof.

**Theorem 4** Given a function  $f$  in  $F$ , if the execution of  $f$  can simulate  $f^1$  under concurrency, then for any  $n \geq 1$ , the execution of  $f$  can simulate  $f^n$  under concurrency.

$$(\exists R. f^1 \sqsubseteq_R f) \rightarrow (\forall n \geq 1. \exists R. f^n \sqsubseteq_R f).$$

Compared to Yggdrasil [66], which assumes that failure happens only once when verifying the idempotence of sequential recovery procedures, Flux targets a different setting — concurrent vs. sequential. Yggdrasil proves that if the execution with one retry produces the same system state and return value as the retry-free execution, then the execution with arbitrary times of retries also produces the same system state and return value. This approach ignores intermediate system states. It only considers the system state and return value when the function finishes because intermediate system states for sequential functions are not observable to clients. However, under concurrency, we should consider intermediate system states. We need to define and prove failure reduction based on simulation relation  $\sqsubseteq_R$ .

## 5 Implementation

Flux builds a verifier to automatically verify idempotence simulation for each function based on failure reduction. It models the execution of a function with symbolic traces generated by

symbolic execution. Each trace represents a feasible execution path and records the path condition, events, and database operations. The verifier can only handle Java applications because Flux builds the verifier by extending a symbolic execution engine for Java [61]. However, our definition and verification method for idempotence consistency is not specific to any particular programming language.

When functions invoke third-party services, Flux mandates that developers explicitly indicate whether these services have side effects. In particular, developers provide a vector of service names and a corresponding bit vector, where each bit indicates whether the corresponding service has side effects. For instance, developers should annotate the *random* function with a “0” since it has no side effects, whereas developers should annotate the *print* function with a “1”.

Next, Flux handles unbounded loops by first converting functions into abstract syntax trees (ASTs) and identifying unbounded loops within them. Then, Flux replaces each unbounded loop that does not modify the shared state with an uninterpreted function. It further identifies all variables modified within the loop and assigns the return value of the uninterpreted function to these variables. For each unbounded loop that modifies the shared state, Flux partitions the code into three parts via ASTs and checks them (Section 4.3).

---

### Algorithm 1: Workflow of the Verifier

---

```

1 Input: A function set  $F$ , a function  $f \in F$ , a string vector
   services of the names of services, and a bit vector bv
   indicating whether each service has side effects.
2 Output: The verification result of  $f$ .
3 Verify( $F$ ,  $f$ , services, bv):
4    $R := \text{GenRelyCond}(F)$ 
5    $T := \text{TracesNoRetry}(f, R, \textit{services}, \textit{bv})$ 
6    $T_r := \text{TracesWithRetry}(f, R, \textit{services}, \textit{bv})$ 
7   foreach  $\langle t_r, pc_r \rangle \in T_r$ :
8     if  $\neg \text{HasSimulatedTrace}(\langle t_r, pc_r \rangle, T)$  then
9       return false
10  return true
11  $\text{HasSimulatedTrace}(\langle t_r, pc_r \rangle, \textit{Traces})$ :
12  foreach  $\langle t, pc \rangle \in \textit{Traces}$ :
13    if  $\text{CheckSimulation}(t_r, pc_r, t, pc)$  then
14      return true
15  return false

```

---

Algorithm 1 shows the verification algorithm. The goal is to prove the simulation relation between  $f^1$  defined in Section 4.4 and  $f$ . First, *GenRelyCond* generates the rely condition  $R$  by symbolically executing all functions in  $F$ . Based on  $R$ , *TracesNoRetry* generates all possible symbolic traces  $T$  for  $f$  via another symbolic execution. *TracesWithRetry* returns all possible symbolic traces  $T_r$  for  $f^1$ . Then, for every trace  $t_r \in T_r$ , *HasSimulatedTrace* checks whether there exists a trace in  $T$  that can simulate  $t_r$ . The initial path condition of symbolic execution is true, which does not contain any constraints on database states and function arguments. Therefore, if Flux can find a retry-free trace for each trace  $t_r$ , the retry-free trace is feasible, and the idempotence simulation holds for any possible database states and arguments.



## 5.1 Generating Symbolic Traces

Each trace is an ordered list with a path condition. Every list element includes the following fields: step id, event, database operation, and the post-condition after the step. The step id identifies the atomic step causing state transition. The event produced by the step has a type, some arguments, and a return value, which are symbolic expressions or constants. Flux considers three types of events for the execution of a serverless function: function invocation, function response, and third-party services with side effects. Developers specify whether a third-party service has side effects via the bit vector  $bv$  in Algorithm 1 (line 3). The fields of the operation include the type ( $optype$ ), the argument ( $oparg$ ), and the result of the operation ( $opret$ ), where  $oparg$  and  $opret$  are symbolic expressions or constants. As described in Section 4.2, Flux models the semantics of each API as a sequence of read or write operations. Post-conditions can help model the return value of read operations. For example,  $D[k]=c$  implies that the results of the subsequent read operations on  $k$  must be  $c$ . Flux adds these propositions about return values of read operations to the path condition.

$GenRelyCond(F)$  symbolically executes all functions in  $F$  and returns the rely condition  $R$ . As illustrated in Section 4.2, Flux identifies three kinds of data. If an operation modifies the data indexed by a symbolic key, Flux assumes that the operation can change arbitrary data in the database. If an operation writes a symbolic value into the data, Flux uses the path condition to infer whether it is a constant or an arbitrary value. Users can also annotate that some variables in a serverless function are unique. That means other concurrent instances cannot access the data indexed by these unique variables. For example, developers can annotate  $receiptId$  in Figure 1 to be unique to indicate that other concurrent instances cannot write the receipt created by the current instance.

## 5.2 Checking Idempotence Simulation

---

### Algorithm 2: Checking Idempotence Simulation

---

```

1  $CheckSimulation(t_r, pc_r, t, pc)$ :
2    $premise := pc_r \wedge pc$ 
3    $pass := CheckWithPremise(t_r, t, premise)$ 
4   return  $pass$ 
5
6  $CheckWithPremise(t_r, t, premise)$ :
7   if  $t_r.empty()$  then
8     return true
9    $step := t_r.subtrace(0, 1)$ 
10  foreach  $n$  from 0 to  $t.size() - 1$ :
11     $nsteps := t.subtrace(0, n)$ 
12     $pass := CheckStep(step, nsteps, premise)$ 
13    if  $!pass$  then
14      continue
15     $next\_premise := UpdatePremise(step, nsteps, premise)$ 
16     $next\_t_r := t_r.subtrace(1, t_r.size())$ 
17     $next\_t := t.subtrace(n, t.size())$ 
18     $pass := CheckWithPremise(next\_t_r, next\_t, next\_premise)$ 
19    if  $pass$  then
20      return true
21  return false

```

---

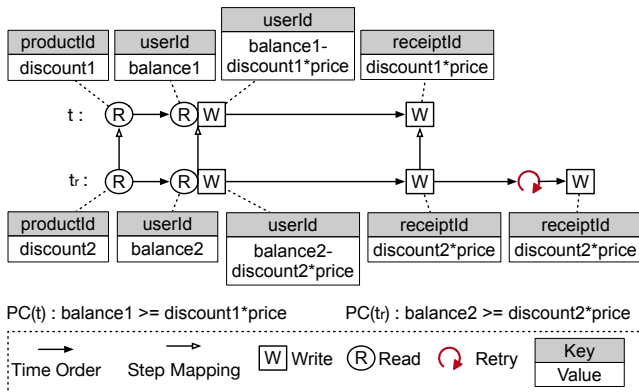
$CheckSimulation$  in Algorithm 2 determines if two traces,

$t_r$  from  $f^1$  and  $t$  from  $f$ , satisfy the idempotence simulation. Their associated path conditions are  $pc_r$  and  $pc$ . Specifically,  $CheckSimulation$  tries to construct a mapping from every step in  $t_r$  to  $n$  ( $n \geq 0$ ) steps in  $t$  such that the  $n$  steps can simulate the single step. The existence of such a step mapping can imply idempotence simulation.

$CheckWithPremise$  recursively checks all possible step mappings. It first uses  $CheckStep$  (Line 12) to check if the  $n$  steps in  $t$  ( $nsteps$ ) can simulate the first step in  $t_r$  ( $step$ ). If the check fails, it increases  $n$  to enumerate other possible mappings. Otherwise, it continues to check the subsequent steps in  $t_r$  in a recursive way (Line 18). To reason the simulation between  $step$  and  $nsteps$ ,  $CheckStep$  requires that the write operations in  $step$  and  $nsteps$  result in the same database state. It means that every write operation in  $nsteps$  should have the same parameters as the write operation in  $step$  under the proposition of  $premise$ , where  $premise$  is the conjunction of the path conditions associated with  $t$  and  $t_r$ . Specifically, two symbolic parameters,  $p_1$  and  $p_2$ , are equivalent under  $premise$  if  $premise \rightarrow (p_1 = p_2)$  is true. Flux leverages an SMT solver to check this first-order logic formula. If  $step$  does not record any write operations,  $nsteps$  should also contain no write operations. Besides, when  $step$  has an event, such as invoking a third-party service with side effects and function response, Flux requires that  $nsteps$  contains the same event. After the check succeeds, Flux updates  $premise$  with  $UpdatePremise$ , which maintains the relations among symbolic variables.  $UpdatePremise$  has three parameters, including  $step$ ,  $nsteps$ , and  $premise$ . If operations in  $step$  and  $nsteps$  read the data indexed by the same key in databases,  $UpdatePremise$  adds a proposition to  $premise$  that these operations return the same value. Otherwise, it does nothing.

Algorithm 2 enumerates all possible mappings, which introduces heavy verification burdens. Flux proposes a heuristic algorithm based on the observation that  $f^1$  and  $f$  are almost the same, except that the platform may re-execute  $f^1$ . Thus, for each step  $s_i^1$  of  $f^1$  before the retry, Flux tries to map it to the  $i$ th step  $s_i$  of  $f$ . For each step  $s_j^1$  of  $f^1$  after the retry, if  $f^1$  has executed  $s_j^1$  before the retry, then Flux maps it to a nop step, a step that does nothing. Otherwise, it maps  $s_j^1$  to a step  $s_k$  in  $f$  such that  $s_k$  can simulate  $s_j^1$ . This only constructs and checks one mapping instead of enumerating all possible mappings, which may miss the correct mapping. If the constructed mapping does not work, Flux will randomly sample other mappings to reduce false positives. However, the method causes no false positives in the evaluation.

**Example.** After logging all operations except for the  $put$  operation, the  $payment$  function will have no idempotence issues. We also need to log  $generateId$ , which always returns the same value on retry. First,  $GenRelyCond(F)$  returns a rely condition that other concurrent function instances can arbitrarily change  $balance$  and  $discount$  in the database because



**Figure 7:** The example of verifying *payment* in Figure 1 when logging all database operations except for *put* (line 9). Variables, such as *discount1*, have symbolic values.  $PC(t)$  is the path condition of the trace  $t$ .

the function writes them by symbolic values. Second, Figure 7 shows that  $t_r$  is a symbolic trace produced by retrying *payment*<sup>1</sup> after it generates *receiptId*, while  $t$  is a symbolic trace produced without retries. Since Flux models *update* (line 4) as a read and a write operation executed atomically, there is no arrow between them. The post-conditions in traces are true. Third, Algorithm 2 finds a proper mapping from every step in  $t_r$  to  $n$  steps in  $t$  and returns true. According to the heuristic algorithm in the previous paragraph, there are three steps in  $t_r$  before the retry. Flux maps each of them to one step in  $t$ , accordingly. For instance, the first step in  $t_r$  corresponds in  $t$ , accordingly. The last step in  $t_r$  corresponds to the *put* (line 9) executed in the first execution of *payment*<sup>1</sup>. Thus, Flux maps this step to a nop step. The first step in  $t$  can simulate the first step in  $t_r$  because they do not modify the database state and produce no events. Since the start database states of the two steps are the same, the return values of their read operations are the same, which means  $discount1 = discount2$ . The second step in  $t$  can simulate the second step in  $t_r$  because they change *balance* to the same value with no events. Due to a similar reason, the third step in  $t$  can simulate the third step in  $t_r$ . Because of logging, *get* on retry in  $t_r$  still returns *discount2* in the first execution, while *update* on retry in  $t_r$  does nothing and returns true. The *generateId* function also returns the same identifier. Flux can use a nop step to simulate the last step in  $t_r$  because it is a useless write that does not change the data in the receipt. Since the function has no return value, we omit the function response event. Other traces of *payment* under retry can also pass the verification. Therefore, *payment* with these logs has no idempotence issues.

**Soundness and Completeness.** The verifier is sound and incomplete. Soundness means the verifier will not overlook any idempotence issues, which the theorems in Section 4 can imply. Incompleteness means some idempotence-consistent

```

1 void unsupportedLoop(key, n) {
2   value := get("Data", key);
3   while(value % n != 0) {
4     put("Data", key, value + 1);
5     value := get("Data", key);
6   }
7 }

```

**Figure 8:** An example of an unbounded loop unsupported by Flux.

applications cannot pass the verification. Note that although the verifier is incomplete, it can still ensure idempotence consistency and reduce the performance overhead of logging. The verifier is incomplete and will introduce false positives in the following cases. First, an application is idempotence-consistent, but individual functions do not satisfy idempotence simulation. For example, an application incorporates two functions,  $f_1$  and  $f_2$ . They blindly write different values to the same record  $R_a$ , while no functions will read it. Consider the interleaving of  $f_1.write(R_a, 1) \rightarrow f_2.write(R_a, 0) \rightarrow f_1.retry() \rightarrow f_1.write(R_a, 1)$ . In this example, functions flip  $R_a$ 's value twice due to the retry. Nevertheless, since no functions read the record  $R_a$ , clients will not observe that two flips occur. Idempotence consistency still holds. However, Flux will consider these two writes as non-idempotent because they fail to follow idempotence simulation and will log each of them; Second, an application contains unbounded loops with write operations such that the parameters of issued write operations can be different between normal execution and retry. For example, the unbounded loop in Figure 8 uses the data read from the database to be the parameters of write operations and the loop condition. As a result, the parameters of write operations and the number of loop iterations may change on retry, which does not satisfy the requirements of Theorem 2. Section 4.3 depicts two specific types of unbounded loops that can be handled by Flux. Third, when examining the idempotence simulation of a function  $f$ , Flux constructs a step mapping using a heuristic approach instead of enumerating all potential mappings. The heuristic runs under the assumption that if  $f$  fails before a specific operation  $op$ ,  $f$  will execute  $op$  on retry. When the assumption does not hold, the heuristic may not work, and Flux may miss correct mappings. Last, Flux does not generate all possible rely, pre- and post-conditions, which may impede the verification capability of idempotence simulation.

## 6 Advisor

Advisor identifies idempotence-violating operations based on a brute-force algorithm and a heuristic algorithm. The brute-force algorithm first enumerates all possible operation sets. Second, for every set, it invokes the verifier to prove the function after ensuring exactly-once execution of each operation in the set via logs. Flux models the exactly-once execution of an operation by adjusting the generated traces rather than modifying functions. Specifically, it deletes each trace element that records a retried write operation in the set.

**Table 1:** An example of executing the heuristic algorithm of advisor.

get	0	1	1
update	1	0	1
put	1	1	0
Verify( $F, f$ )	False	False	True

Furthermore, it appends a formula to the path condition for each retried read operation in the set, indicating that the retried operation returns the same result as the initial execution. Third, among all operation sets that enable the function to pass the verification, Flux selects the set that incurs the least performance cost. Flux estimates the performance cost via the evaluation results derived from the adopted logging mechanism. This algorithm enumerates  $O(2^n)$  possible operation sets, where  $n$  is the number of operations.

To reduce the complexity, we propose a heuristic algorithm checking only  $O(n)$  sets. Initially, Flux logs all operations in the function. Then Flux gradually removes the logs of operations one by one. If the function can pass the verification after removing a log, advisor will permanently remove the log. Otherwise, the operation is idempotence-violating, and advisor preserves the log. Table 1 shows how to apply the heuristic to *payment* in Figure 1. Assume that advisor has logged *generateId*. “0” denotes that advisor logs the operation. “1” denotes that advisor does not log the operation. Advisor first removes the log of *get* ( $get=0$ ) but  $Verify(F, f)$  returns *false*. Therefore, advisor preserves the log for idempotence. Due to the same reason, advisor does not remove the log of *update*. Last,  $Verify(F, f)$  indicates removing the log of *put* does not break the idempotence. Therefore, advisor does not log it.

The incompleteness of the verifier may result in advisor being unable to detect certain redundant logs. Nonetheless, Flux can still diminish logging overhead for functions while ensuring idempotence consistency. Specifically, for an unbounded loop that the verifier cannot handle, advisor logs all operations before and within the loop except for read operations on read-only data. Then advisor directly addresses the code after the unbounded loop.

## 7 Evaluation

We aim to answer the following questions: 1) How effective is the verifier? 2) How long does it take for advisor to identify idempotence-violating operations? 3) How much performance benefit does Flux bring?

### 7.1 Experimental Setup

We evaluate the execution time of the verifier and advisor on a desktop running Ubuntu 18.04, which has an Intel Core i7-8700 processor and 15GB DRAM. Additionally, we evaluate the performance of serverless applications on multiple AWS servers.

We compare our system with Beldi [73] and Boki [45],

which logs all operations. Although some other systems [44, 69] also guarantee idempotence for serverless applications, we focus on comparing with Beldi and Boki because they are state-of-the-art. It is worth noting that while Flux aims to provide idempotence consistency, Beldi and Boki go further by ensuring transactional properties as well. To ensure fairness in our comparison, we do not utilize the transaction mechanisms provided by Beldi and Boki, as our focus is on idempotence assurance alone. Therefore, the advantage that Flux may have over Beldi and Boki in terms of performance does not result from its lack of guarantee for transactional properties.

We store the data of applications in DynamoDB [15]. We run Beldi on AWS Lambda [16] with 1GB DRAM for each instance and collect performance results via AWS CloudWatch. Boki provides its own serverless platform. We deploy Boki according to the evaluation environment in its paper [45]. Boki’s serverless platform can report the latencies of functions but cannot report their throughputs precisely.

We use wrk2 [1] as the load generator, which runs on an m5.2xlarge instance for Beldi and a c5d.xlarge instance for Boki. We adapt representative applications from the AWS serverless applications repository [3], a GitHub repository (10.9k stars) [9], popular benchmarks [8, 10, 29], and applications commonly used in papers about serverless computing [5, 6]. They have covered diverse real-world scenarios of serverless computing, such as image processing and web applications. We choose the applications with at least 1,000 deployments in AWS serverless application repository [3]. We skip some applications since they are micro-benchmarks or stateless (no database operations). Table 2 summarizes the characteristics of these applications. **Type-I** applications satisfy idempotence consistency, while **Type-II** applications do not. Although most applications in Table 2 have fewer than a thousand lines of code, they remain representative because serverless platforms typically impose restrictions on code size and running time [2, 4, 7]. For instance, existing work [75] that adapts web applications to serverless platforms needs to largely reduce the application’s code size.

Flux’s approach is orthogonal to specific programming languages. However, its implementation depends on a Java symbolic execution engine [61] for program analysis. We manually port applications to Go with the same semantics for a fair performance comparison with Beldi and Boki, which target Go applications. We choose to implement a verifier for applications in Java rather than Go because Java is a more frequently utilized language in developing serverless applications than Go [11].

### 7.2 Verification Efficacy

Table 2 shows that the verifier identifies all 12 applications with idempotence issues. All issues are previously unknown. Developers have confirmed 8 issues among them. The verifier works for all applications except SPECjbb, which has

**Table 2:** The characteristics of 27 serverless applications. The applications with † have unbounded loops. **LoC** indicates lines of Java code. **C/S** indicates whether functions run sequentially (S) or concurrently (C). **#F**, **#I**, and **#N** indicate the number of functions, functions without idempotence issues, and functions with issues. **#R/#W** and **#S** indicate the number of read/write operations and idempotence-violating operations. **VTime**, **ATime(H)**, and **ATime(B)** indicate the execution time (in seconds) of the verifier, advisor using the heuristic algorithm, and advisor using the brute-force algorithm.

Type	Application	LoC	C/S	Selected	#F	#I	#N	#R	#W	#S	VTime	ATime(H)	ATime(B)
I (15)	Data Analysis [10] †	356	S	✓	7	7	0	4	3	0	5.45		
	Image-Processing [10]	435	S		5	5	0	0	4	0	3.24		
	Mapreduce [8] †	250	S		3	3	0	3	1	0	2.06		
	FaaSImage [8]	193	S		1	1	0	1	9	0	104.31		
	Video [8] †	40	S		1	1	0	1	1	0	2.19		
	Image-Resizer [9]	92	S		1	1	0	1	1	0	2.40		
	Replicator [9]	59	S	✓	1	1	0	1	1	0	2.40		
	Receive-Email-Body [9]	58	S		1	1	0	1	0	0	0.74		
	Fetch-And-Store [9]	66	S		1	1	0	0	1	0	1.49		
	FFmpeg [9] †	49	S		1	1	0	1	-	-	2.23		
	DynamoDB-backup [9] †	26	S		1	1	0	1	-	-	2.44		
	Lambda-Image-Resizer [3]	123	S		1	1	0	1	0	0	1.12		
	Uploader [3]	101	S	✓	1	1	0	2	1	0	1.95		
	FFmpeg-Lambda-Layer [3]	86	S		1	1	0	1	1	0	2.30		
Image-magick [3]	86	S		1	1	0	1	1	0	2.49			
II (12)	SPECjbb2015 [29] †	1,861	C	✓	9	5	4	-	-	-	Timeout	6.23	11.87
	Alexa [10] †	89	C	✓	10	9	1	1	1	1	2.13	2.60	4.25
	Hotel [5] †	714	C	✓	10	8	2	7	5	3	2.26	17.70	68.94
	Media [5]	486	C		7	6	1	1	7	7	14.79	86.04	344.77
	Pynamodb-S3-URL [9]	224	C		6	2	4	6	9	9	9.91	22.85	69.71
	Rest-API [9]	135	C		4	1	3	2	2	3	4.90	5.98	23.72
	GraphQL [9]	80	C	✓	1	0	1	1	1	1	2.07	4.34	12.17
	Mongodb-Atlas [9]	83	S		1	0	1	0	1	1	2.28	14.25	72.67
	Express [9]	76	C		1	0	1	1	1	1	1.90	1.92	4.28
	Flask [9]	34	C		1	0	1	1	1	1	2.31	2.80	3.64
	Save [3]	62	S		1	0	1	0	1	1	2.14	5.61	12.57
	HttpEP [3]	105	C	✓	1	0	1	1	3	3	2.42	11.73	44.5

unbounded loops unsupported by the verifier. However, all functions in SPECjbb that cannot be verified have idempotence issues. Thus, the verifier does not introduce false positives for SPESjbb. Nonetheless, it is worth noting that false positives are still possible for other applications.

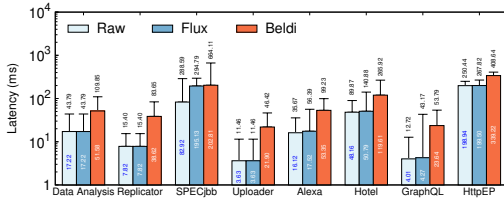
The 21 non-idempotent functions detected by Flux in 12 applications result in various bug patterns and outcomes due to incorrectly repeated write operations on retry. First, the database state is inconsistent with user expectations. For instance, the idempotence violation in SPECjbb causes duplicate balance deductions. Second, the value responded to clients is inconsistent with the database state. An example is an IoT application called Alexa, where a function successfully modifies device configuration but returns “failed” instead of “success” to clients. Third, a single write operation may update multiple records on retry, resulting in duplicated records with identical content. For example, the *PlaceOrder* function in the Hotel benchmark always places a new order with a random identifier on each retry, resulting in duplicate records. Last, concurrent functions may observe the inconsistent shared state. An example is the Media benchmark, which relies on a counter in the database to perform synchronization among concurrent functions. However, the *ComposeReview* function will falsely increase the counter due to retry, leading to false synchronization among concurrent instances.

To test the scalability of the verifier, we run micro-benchmarks with increasing verification complexity. When the number of branches in a single function increases, the verification time increases exponentially, as the number of traces also increases exponentially. Verifying a function with 16 branches takes 1441.48 seconds. When we increase the number of database operations, functions, or LoC, the verification time increases linearly because the number of generated traces increases linearly. Figure 15 in Appendix D presents more details. Note that the verification time does not affect the execution time of applications.

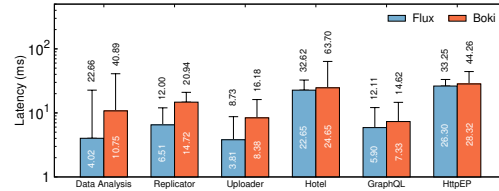
### 7.3 Performance of Advisor

For the second question, Table 2 shows the execution time of advisor with two different algorithms. Compared to the brute-force algorithm, the heuristic algorithm achieves up to 4× smaller search space, which cuts down 80.39% of the execution time of advisor. Although the heuristic algorithm does not guarantee finding the minimum operation set, the evaluation shows that it finds the same set as the brute-force algorithm in practice. Although SPECjbb2015 has unbounded loops unsupported by the verifier, advisor can still handle it with the method described in Section 6.

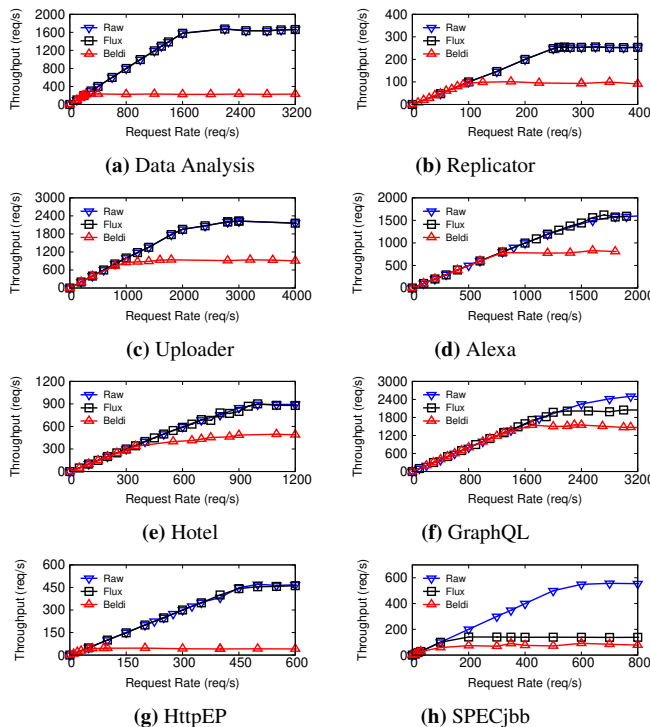




**Figure 9:** The median (box) and 99% tail latency (whisker) of Raw, Flux, and Beldi for eight applications. The y-axis is in the log scale.



**Figure 11:** The median (box) and 99% tail latency (whisker) of Flux and Boki for six applications at 1 RPS.



**Figure 10:** The throughput of each serverless application under different configurations with increasing request rates.

## 7.4 Performance of the Applications

To answer the third question, we run applications under four configurations: Raw, Flux, Boki, and Beldi. Raw means running applications without any logs, which may be non-idempotent. Flux uses existing mechanisms to log operations identified by advisor. Wrk2 runs for 7 minutes to generate random requests. We will show the results of 8 representative applications marked by ✓ in Table 2. Since Type-I applications are idempotence-consistent, Flux removes all logging operations. For Type-II applications, Flux reduces up to 99.47% of logging operations during execution compared to Beldi and Boki.

### 7.4.1 Flux vs. Beldi

**Latency.** Figure 9 shows the results on AWS Lambda. Flux poses no logging overhead over Raw for Type-I applications. Compared with Beldi, which logs all operations, Flux brings 2.5× ~ 6× performance improvement. For Type-II applications, Flux can avoid logging some operations. As a result, it

achieves up to 5.5× performance improvement over Beldi.

**Throughput.** Figure 10 shows that Flux achieves the same peak throughput on Type-I applications as Raw and up to 7.36× higher peak throughput than Beldi. For Type-II applications, Flux can avoid part of logging operations. Therefore, it has up to 80% higher peak throughput than Beldi, except for HttpEP. For HttpEP, Flux has 10× higher peak throughput than Beldi because Flux avoids logging a scan operation that can return massive data, which reduces much overhead under concurrency.

### 7.4.2 Flux vs. Boki

Boki [45] is another system to provide idempotence for serverless applications. The comparison can also demonstrate that Flux is general enough to be independent on a specific serverless platform. Because of Boki’s limitations, it does not guarantee idempotence for read-modify-write database operations [13]. Thus, we do not evaluate Alexa and SPECjbb on Boki.

Figure 11 shows that at 1 RPS, Flux reduces the median latency by up to 62.6% compared to Boki. The performance of Boki is better than Beldi because Boki designs a more efficient logging mechanism. Under high concurrency, the logging mechanism of Boki introduces more overhead. We further evaluate the latency at 800 RPS. Flux reduces the median latency by up to 82.5% and the p99 latency by up to 69.0% for Type-I applications compared to Boki because Flux introduces no logs. For Type-II applications, Flux reduces the median latency by up to 88.4% and the p99 latency by up to 72.4%. Flux achieves 8.7× lower median latency than Boki on HttpEP because Flux avoids logging an expensive “scan”.

### 7.4.3 Performance of the Java Applications

Since Beldi and Boki only support Go applications, we compare the performance of Java applications on AWS Lambda under the configurations of Flux and Raw. We implement the logging mechanism via transactions [73]. At 1 RPS, Flux achieves a tail latency up to 22.6% higher than Raw due to the additional logging overhead. However, despite better performance, Raw cannot guarantee idempotence consistency and may cause incorrect execution results.

## 8 Related Work

**Verification of Idempotence.** Table 3 summarizes the major differences between Flux and prior works that can verify

**Table 3:** Main differences between Flux and prior works. FSCQ-based works use a method akin to FSCQ [28] to prove the idempotence. Partial protection means ensuring idempotence consistency while reducing unnecessary logs.

	Definition of Idempotence		Verification Method of Idempotence				Protection
	Support Concurrency	Target Serverless Applications	Verification Targets	Support Concurrency	Automated Verification	Unbounded Loop	Partial Protection
Flux	✓	✓	Implementation	✓	✓	✓(Partially)	✓
Ramalingam et al. [62]	✓	✗	Protocol	✓	✗	–	✗
Jangda et al. [44]	✓	✓	Protocol	✓	✗	–	✗
Yggdrasil [66]	✗	✗	Implementation	✗	✓	✗	–
FSCQ-Based Works [23–25, 27, 28]	✗	✗	Implementation	✗	✗	✓	–

idempotence. Jangda et al. [44] formalizes the semantics of serverless computing and ensures idempotence with transactions. There are two main differences between Jangda et al. and Flux. For the idempotence definition, idempotence consistency is more relaxed than the idempotence provided by Jangda et al., as it allows more concurrent schedulings. Jangda et al. tries to model serverless computing with naive semantics to conceal the low-level details of serverless function execution, such as concurrency and warm-start. It requires the platform to process concurrent requests in the same way as processing a single request at a time without concurrency or retries. Consequently, it necessitates atomicity, serializability, and exactly-once execution. Besides, Jangda et al. focuses on verifying protocols instead of source code. To ensure idempotence, it protects the entire function with serializable transactions and uses logs to ensure that the transaction only commits once. However, using transactions is not optimal since some applications may not require transaction semantics, resulting in redundant protection and performance cost.

FSCQ [28] and DFSCQ [27] verify the crash safety of file systems via Crash Hoare Logic (CHL). CHL requires developers to manually write pre-conditions, post-conditions, and crash conditions to specify crash safety. Under crash, CHL proves that the program state always satisfies crash conditions after a crash happens at any time. It defines the idempotence of a recovery program to be that crash conditions imply pre-conditions. Perennial [23], GoJournal [24], and DaisyNFS [25] achieve significant progress in the verification of concurrent crash-safe systems. They adopt a similar approach as FSCQ to verify the idempotence of a recovery program. Thus, they cannot realize automated verification. Different from them, Flux focuses on automated verification without human effort.

Recent SMT-based verification approaches [26,39,66] have solved many issues of the automated verification of storage systems. Yggdrasil proposes a new correctness definition for sequential file systems — *crash refinement*. It means that for any disk state produced by the implementation with crash recovery, the specification can also produce the same disk state. The definition is amenable to automated verification. Yggdrasil also verifies the idempotence of recovery functions.

Unfortunately, it verifies sequential functions rather than concurrent functions. Others [26,39] also verify sequential functions.

Ramalingam et al. [62] formally defines idempotence in the general distributed setting, proving that logging each operation can guarantee this property. It has two main differences from Flux. First, for the definition of idempotence, both Ramalingam et al. and Flux require that any execution with retries has the same observable behavior as another normal execution. However, Ramalingam et al. targets a general distributed setting. The observable behavior only includes function invocation and response. In contrast, Flux also considers database states. As a result, for stateful serverless applications, Ramalingam et al.’s definition may allow inconsistent database states on retry as long as functions do not return database states to clients. This anomaly is critical if clients directly access database states for analysis or other purposes. In contrast, Flux’s definition prohibits inconsistent database states and can prevent such anomalies. Second, for verification and protection, Ramalingam et al. manually proves the protocol of logging each operation and presents a compiler that automatically logs each operation. Flux focuses on verifying the implementation of applications and its advisor only logs necessary operations.

**Crash-Only Software.** Previous work [20] comprehensively analyzes the requirements for crash-only software, a program that can crash safely and recover quickly via retries. Our work can assist in revising these requirements. Specifically, serverless applications fulfill most of these requirements, such as explicit boundaries around what is retried and dedicated storage for non-volatile data. However, our work only mandates that all functions satisfy idempotence consistency instead of necessitating that every request is idempotent, as stated in previous work [20]. This distinction results in fundamental differences in several aspects. First, idempotence pertains to a single component, while idempotence consistency pertains to the entire application comprised of multiple components. Second, while idempotence requires a function’s execution result to remain the same despite retries, idempotence consistency only requires the existence of an execution without retries for each execution with retries such that they produce the same result. For example, a read-only function

may not satisfy idempotence because concurrent modifications to the database state can result in different return values with and without retries. However, read-only functions do not violate idempotence consistency. In addition to serverless functions, other applications with “imperfectly crash-only” code can also refer to the requirements revised based on Flux.

**Research on Serverless Computing.** Some systems [44, 45, 62, 69, 73] focus on ensuring idempotence via runtime mechanisms. Different from them, Flux focuses on verifying idempotence consistency. Furthermore, combining them with Flux can ensure idempotence efficiently. Some other systems [21, 30–32, 46, 49, 51, 56, 63, 65, 71, 72, 74] target other things, such as startup time.

**Consistency Model.** There are many consistency models [14], which define the permitted execution order under concurrency. Idempotence consistency specifies the expected behavior of concurrent systems when failure happens, which is the main difference from existing consistency models.

## 9 Discussion

**Serverless Applications vs. Other Applications.** *Idempotence* [43] is a property that is important not only for serverless applications but also for programs that use retry-based fault tolerance approaches, such as RPC-based distributed systems [41, 50], AI systems [38], and even some intermittent systems [70]. Thus, it is worth considering *whether we can apply Flux to programs beyond serverless applications*. The answer is *yes and no*.

One of the key contributions of Flux is formally defining idempotence consistency, which is general enough for various scenarios. For instance, different RPC handlers in distributed systems may concurrently manipulate shared states. Repeated state updates due to failures and retries could impair data consistency under concurrency. Specifically, we successfully detect an issue in HDFS according to the definition of idempotence consistency, which the community has confirmed [41]. When the system retries NameNode RPC of *ClientProtocol.truncate*, it may truncate a file multiple times, which will potentially cause data loss if another RPC simultaneously updates the same file. However, this issue will not happen under sequential execution.

However, using Flux’s method to verify other systems poses many challenges. First, the automated verification algorithm mainly focuses on serverless applications. We need to redesign it for other scenarios. For example, Flux presumes that shared states are solely key-value pairs stored in NoSQL databases. However, shared states could be file descriptors, shared variables, or global configurations in distributed systems. Programmers must reinterpret them across various scenarios. Additionally, Flux automatically constructs pre-, post-, and rely conditions based on the NoSQL interface assumption, necessitating a re-examination of the construction algorithm based on other modeling methods of states. Second, Flux’s

implementation targets serverless applications. For example, Flux currently only supports Java programs and DynamoDB, which are commonly used by serverless applications. Engineering effort is necessary to support other languages and storage services. Furthermore, advisor in Flux relies on existing logging mechanisms for serverless applications to fix idempotence issues. Therefore, the new scenario should also provide mechanisms to ensure exactly-once execution of operations on shared states. Failure to do so limits Flux’s potential to identify and fix idempotence issues via advisor.

**Idempotence Consistency vs. Atomicity.** Although atomicity is vital for fault tolerance, Flux does not guarantee it. Atomicity is not mandatory for some applications to sustain fault tolerance, as they may permit other functions to see partial updates. Furthermore, it is essential to emphasize that Flux is orthogonal to approaches that ensure atomicity. An application that necessitates atomicity can still utilize Flux to guarantee idempotence consistency and minimize logging overhead.

Although Flux does not verify atomicity and some other transactional properties, extending our approach to verify these properties is an intriguing research direction. Some verifiers [22–25, 37, 53, 54, 76] target transactional properties but may not consider retries. After ensuring idempotence consistency based on Flux, we can prove the transactional properties of functions without considering failures and retries. Therefore, we can combine Flux and these verifiers by ensuring idempotence consistency via Flux and then proving other properties with these verifiers.

**Automated Verification vs. Interactive Verification.** We adopt symbolic execution engines to develop Flux, which requires less manual effort than using interactive theorem provers. [37]. Additionally, although interactive theorem provers can handle unbounded loops by crafting loop invariants, finding proper loop invariants is notoriously difficult.

## 10 Conclusion

This paper presents Flux, the first toolkit that can automatically verify and help ensure the idempotence consistency of serverless applications. It guarantees idempotence consistency via logs while reducing unnecessary logging overhead.

## Acknowledgment

We sincerely thank the anonymous reviewers for their valuable comments. We are especially grateful to our shepherd, George Candea, whose reviews and suggestions largely improved our work. This work is supported by the National Natural Science Foundation of China (No. 62132014 and 62272304), the Fundamental Research Funds for the Central Universities, and the HighTech Support Program from Shanghai Committee of Science and Technology (No. 20ZR1428100). Zhaoguo Wang ([zhaoguowang@sjtu.edu.cn](mailto:zhaoguowang@sjtu.edu.cn)) is the corresponding author.

## References

- [1] A Constant Throughput, Correct Latency Recording Variant of wrk. <https://github.com/giltene/wrk2>.
- [2] AWS Lambda Enables Functions That Can Run up to 15 minutes. [https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/?nc1=h\\_ls](https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/?nc1=h_ls).
- [3] AWS Serverless Application Repository. <https://serverlessrepo.aws.amazon.com/applications>.
- [4] Azure Functions Hosting Options. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale>.
- [5] Beldi. <https://github.com/eniac/Beldi>.
- [6] Benchmark Workloads of Boki. <https://github.com/ut-osa/boki-benchmarks>.
- [7] Cloud Functions Execution Environment. <https://cloud.google.com/functions/docs/concepts/execution>.
- [8] Functionbench. <https://github.com/kmu-bigdata/serverless-faas-workbench>.
- [9] Serverless Examples. <https://github.com/serverless/examples>.
- [10] Serverlessbench. <https://serverlessbench.systems/en-us/>.
- [11] State of Serverless. <https://www.datadoghq.com/state-of-serverless/>.
- [12] Uninterpreted Functions and Constants. <https://microsoft.github.io/z3guide/docs/logic/Uninterpreted-functions-and-constants>.
- [13] Working with Items and Attributes - Amazon DynamoDB. <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.AtomicCounters>.
- [14] Marcos K. Aguilera and Douglas B. Terry. The Many Faces of Consistency. *IEEE Data Eng. Bull.*, 39:3–13, 2016.
- [15] Amazon. AWS Dynamodb. <https://aws.amazon.com/dynamodb/>.
- [16] Amazon. AWS Lambda. <https://aws.amazon.com/lambda/>.
- [17] Amazon. Build a CRUD API with Lambda and DynamoDB. <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamodb.html>.
- [18] Amazon. Make a Lambda Function Idempotent. <https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/>.
- [19] Andreas Blass and Yuri Gurevich. Inadequacy of Computable Loop Invariants. *ACM Trans. Comput. Logic*, 2(1):1–11, jan 2001.
- [20] George Candea and Armando Fox. Crash-Only Software. In *9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003. USENIX Association.
- [21] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 58–64, New York, NY, USA, 2021. Association for Computing Machinery.
- [22] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nikolai Zeldovich. Verifying Concurrent Software Using Movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 306–322, Carlsbad, CA, October 2018. USENIX Association.
- [23] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Huntsville, ON, Canada, October 2019.
- [24] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nikolai Zeldovich. GoJournal: A Verified, Concurrent, Crash-safe Journaling System. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pages 423–439. USENIX Association, July 2021.
- [25] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying the DaisyNFS Concurrent and Crash-safe File System With Sequential Reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.
- [26] Yun-Sheng Chang, Yao Hsiao, Tzu-Chi Lin, Che-Wei Tsao, Chun-Feng Wu, Yuan-Hao Chang, Hsiang-Shang Ko, and Yu-Fang Chen. Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer. In *14th USENIX Symposium on Operat-*



- ing *Systems Design and Implementation*, pages 81–97. USENIX Association, November 2020.
- [27] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286, New York, NY, USA, 2017. Association for Computing Machinery.
- [28] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nikolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, New York, NY, USA, 2015. Association for Computing Machinery.
- [29] Standard Performance Evaluation Corporation. Specjbb 2015. <https://www.spec.org/jbb2015/>.
- [30] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–813, New York, NY, USA, 2022. Association for Computing Machinery.
- [31] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, New York, NY, USA, 2020. Association for Computing Machinery.
- [32] Alexander Fuerst and Prateek Sharma. *FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching*, pages 386–400. Association for Computing Machinery, New York, NY, USA, 2021.
- [33] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop Invariants: Analysis, Classification, and Examples. *ACM Comput. Surv.*, 46(3), jan 2014.
- [34] Google. Google Cloud Functions. <https://cloud.google.com/functions/>.
- [35] Google. Retrying Event-Driven Functions. <https://cloud.google.com/functions/docs/bestpractices/retries>.
- [36] Google. Stateful Serverless on Google Cloud with Cloudstate and Akka Serverless. <https://cloud.google.com/blog/topics/developers-practitioners/stateful-serverless-on-google-cloud-with-cloudstate-and-akka-serverless>.
- [37] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 653–669, Savannah, GA, November 2016. USENIX Association.
- [38] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.
- [39] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 99–115. USENIX Association, November 2020.
- [40] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, New York, NY, USA, 2015. Association for Computing Machinery.
- [41] Apache HDFS. HDFS-16322. <https://issues.apache.org/jira/browse/HDFS-16322>.
- [42] Hadoop HDFS. HDFS-7926. <https://issues.apache.org/jira/browse/HDFS-7926>.
- [43] Pat Helland. Idempotence Is Not a Medical Condition: An Essential Property for Reliable Systems. *Queue*, 10(4):30–46, apr 2012.
- [44] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.*, 3, October 2019.
- [45] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, New York, NY, USA, 2021. Association for Computing Machinery.
- [46] Zhipeng Jia and Emmett Witchel. *Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices*, pages 152–166. Association for Computing Machinery, New York, NY, USA, 2021.
- [47] C. B. Jones. Tentative Steps toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983.

- [48] Kafka. Kafka-5169. <https://issues.apache.org/jira/browse/KAFKA-5169>.
- [49] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference*, pages 805–820. USENIX Association, July 2021.
- [50] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. *Implementing Linearizability at Large Scale and Low Latency*, pages 71–86. Association for Computing Machinery, New York, NY, USA, 2015.
- [51] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaasFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, New York, NY, USA, 2022. Association for Computing Machinery.
- [52] Hongjin Liang and Xinyu Feng. Modular Verification of Linearizability with Non-Fixed Linearization Points. *SIGPLAN Not.*, 48(6):459–470, jun 2013.
- [53] Hongjin Liang, Xinyu Feng, and Ming Fu. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 455–468, New York, NY, USA, 2012. Association for Computing Machinery.
- [54] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–210, New York, NY, USA, 2020. Association for Computing Machinery.
- [55] Nancy Lynch and Frits Vaandrager. Forward and Backward Simulations. *Inf. Comput.*, 128(1):1–25, jul 1996.
- [56] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. Orion and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.
- [57] Microsoft. Designing Azure Functions for Identical Input. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-idempotent>.
- [58] Microsoft. Microsoft Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [59] Microsoft. What are Durable Functions. <https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp>.
- [60] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, apr 2007.
- [61] Java Pathfinder. Java Pathfinder. <https://github.com/javapathfinder/>.
- [62] Ganesan Ramalingam and Kapil Vaswani. Fault Tolerance via Idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 249–262, New York, NY, USA, 2013. Association for Computing Machinery.
- [63] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Ice-Breaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, New York, NY, USA, 2022. Association for Computing Machinery.
- [64] Serverlessbench. Issue with Exactly-once Execution Semantic of Alexa. <https://github.com/SJTU-IADS/ServerlessBench/issues/6>.
- [65] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-level post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.
- [66] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Savannah, GA, November 2016. USENIX Association.
- [67] Spark. Spark-6133. <https://issues.apache.org/jira/browse/SPARK-6133>.
- [68] Spree. Spree. <https://spreecommerce.org/>.

- [69] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, New York, NY, USA, 2020. Association for Computing Machinery.
- [70] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/O Dependent Idempotence Bugs in Intermittent Systems. *Proc. ACM Program. Lang.*, 3, oct 2019.
- [71] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, New York, NY, USA, 2021. Association for Computing Machinery.
- [72] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. InFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, New York, NY, USA, 2022. Association for Computing Machinery.
- [73] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-Tolerant and Transactional Stateful Serverless Workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 1187–1204. USENIX Association, November 2020.
- [74] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, New York, NY, USA, 2021. Association for Computing Machinery.
- [75] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. BeeHive: Sub-Second Elasticity for Web Services with Semi-FaaS Execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 74–87, New York, NY, USA, 2023. Association for Computing Machinery.
- [76] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In

*Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

## Appendix A Proof of Theorem 1

In this section, we prove that it is sufficient to verify the idempotence consistency of a function set  $F$  by proving that each function  $f \in F$  satisfies idempotence simulation, which is Theorem 1 in the paper. Since our verification approach adopts existing compositional proof techniques in RGSim [53], our formal proof is similar to the proof in that paper.

Let's start by introducing some important concepts. When we have two functions running concurrently, we can consider them as one unit and denote their local state using the symbol  $\sigma_1 \parallel \sigma_2$ , where  $\sigma_1$  represents the local state of the first function and  $\sigma_2$  represents the local state of the second function. Using this notation, we can represent the overall system state as  $\langle \sigma_1 \parallel \sigma_2, D \rangle$  when two functions run concurrently. We use  $\langle \sigma_1 \parallel \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_1 \parallel \sigma'_2, D' \rangle$  to denote that the system can take one step and change the system state from  $\langle \sigma_1 \parallel \sigma_2, D \rangle$  to  $\langle \sigma'_1 \parallel \sigma'_2, D' \rangle$ , producing an event  $\alpha$  (if any). During each execution cycle, the system can either advance the first function by one step or advance the second function by one step. It means that if there exists  $\sigma'_1$  and  $D'$  such that  $\langle \sigma_1, D \rangle \xrightarrow{\alpha} \langle \sigma'_1, D' \rangle$ , then we can imply that  $\langle \sigma_1 \parallel \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_1 \parallel \sigma_2, D' \rangle$ . Similarly, if there exists  $\sigma'_2$  and  $D'$  such that  $\langle \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_2, D' \rangle$ , then we can imply that  $\langle \sigma_1 \parallel \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma_1 \parallel \sigma'_2, D' \rangle$ .

$\langle \sigma_1 \parallel \sigma_2, D \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D \rangle$  denotes that the concurrent execution of two functions with the local state  $\Sigma_1$  and  $\Sigma_2$  can simulate the concurrent execution of another two functions with the local state  $\sigma_1$  and  $\sigma_2$  under the rely condition  $R$ . Here is the detailed definition of the simulation relation. We use  $\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle$  to denote that the system can take  $n$  ( $0 \leq n$ ) steps to change the system state from  $\langle \Sigma_1, D \rangle$  to  $\langle \Sigma'_1, D' \rangle$ , producing an event  $\alpha$  (if any).

- For any local state  $\sigma'_1$  and database state  $D'$ , if  $\langle \sigma_1, D \rangle \xrightarrow{\alpha} \langle \sigma'_1, D' \rangle$ , then there exists a local state  $\Sigma'_1$  such that  $\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle$ . If  $\alpha$  is a *response* event, then  $\langle \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma_2, D' \rangle$ . Otherwise,  $\langle \sigma_1 \parallel \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D' \rangle$ . This condition represents the requirement for the simulation relation when the system advances the first function by one step.
- For any local state  $\sigma'_2$  and database state  $D'$ , if  $\langle \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_2, D' \rangle$ , then there exists a local state  $\Sigma'_2$  such that  $\langle \Sigma_2, D \rangle \xrightarrow{\alpha} \langle \Sigma'_2, D' \rangle$ . If  $\alpha$  is a *response* event, then  $\langle \sigma_1, D' \rangle \sqsubseteq_R \langle \Sigma_1, D' \rangle$ . Otherwise,  $\langle \sigma_1 \parallel \sigma'_2, D' \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D' \rangle$ . This condition represents the requirement for the simulation relation when the system advances the first function by one step, which is similar to the first requirement.
- If  $(D, D') \in R$ , then  $\langle \sigma_1 \parallel \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D' \rangle$ .

We can extend the above definition to more than two functions.

To prove Theorem 1, we first prove the following two lemmas based on the above definitions.

**Lemma 1** Assume there are four functions  $f_1, f_2, g_1$ , and  $g_2$ . Suppose that the execution of  $g_1$  can simulate the execution of  $f_1$  under the rely condition  $R$ , while the execution of  $g_2$  can simulate the execution of  $f_2$  under the rely condition  $R$ . Then we can imply that the concurrent execution of  $g_1$  and  $g_2$  can simulate the concurrent execution of  $f_1$  and  $f_2$  under the rely condition  $R$ . We use  $\sigma_1, \sigma_2, \Sigma_1$ , and  $\Sigma_2$  to represent the local states of  $f_1, f_2, g_1$ , and  $g_2$ , respectively.

$$\begin{aligned} & \forall \sigma_1, \Sigma_1, \sigma_2, \Sigma_2, D. \\ & ((\langle \sigma_1, D \rangle \sqsubseteq_R \langle \Sigma_1, D \rangle) \wedge (\langle \sigma_2, D \rangle \sqsubseteq_R \langle \Sigma_2, D \rangle)) \rightarrow \\ & (\langle \sigma_1 \parallel \sigma_2, D \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D \rangle) \end{aligned}$$

### Proof

The premises include

$$\langle \sigma_1, D \rangle \sqsubseteq_R \langle \Sigma_1, D \rangle, \quad (1)$$

and

$$\langle \sigma_2, D \rangle \sqsubseteq_R \langle \Sigma_2, D \rangle. \quad (2)$$

The conclusion is

$$\langle \sigma_1 \parallel \sigma_2, D \rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D \rangle. \quad (3)$$

Below we prove the conclusion by *co-induction* on  $\sqsubseteq_R$ . According to the definition of  $\sqsubseteq_R$  described above, the execution of two functions belongs to one of the following five cases.

- $\langle \sigma_1, D \rangle \xrightarrow{\alpha} \langle \sigma'_1, D' \rangle$ , where  $\alpha$  is not a *response* event.

According to the definition of  $\sqsubseteq_R$ , we need to prove that there exists  $\Sigma'_1$  such that  $\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle$  and  $\langle \sigma'_1 \parallel \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma'_1 \parallel \Sigma_2, D' \rangle$ .

From Equation (1), there exists  $\Sigma'_1$  such that

$$\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle, \quad (4)$$

and

$$\langle \sigma'_1, D' \rangle \sqsubseteq_R \langle \Sigma'_1, D' \rangle. \quad (5)$$

Since  $(D, D') \in R$ , from Equation (2), we know

$$\langle \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma_2, D' \rangle. \quad (6)$$

From both Equation (5) and Equation (6) we know

$$\langle \sigma'_1 \parallel \sigma_2, D' \rangle \sqsubseteq_R \langle \Sigma'_1 \parallel \Sigma_2, D' \rangle. \quad (7)$$

- $\langle \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_2, D' \rangle$ , where  $\alpha$  is not a *response* event. The proof is similar to the first case.



- $\langle \sigma_1, D \rangle \xrightarrow{\alpha} \langle \sigma'_1, D' \rangle$ , where  $\alpha$  is a *response* event. According to the definition of  $\Xi_R$ , we need to prove that there exists  $\Sigma'_1$  such that  $\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle$ , and  $\langle \sigma_2, D' \rangle \Xi_R \langle \Sigma_2, D' \rangle$ .

From Equation (1), there exists  $\Sigma'_1$  such that

$$\langle \Sigma_1, D \rangle \xrightarrow{\alpha} \langle \Sigma'_1, D' \rangle. \quad (8)$$

Since  $(D, D') \in R$ , we know

$$\langle \sigma_2, D' \rangle \Xi_R \langle \Sigma_2, D' \rangle. \quad (9)$$

- $\langle \sigma_2, D \rangle \xrightarrow{\alpha} \langle \sigma'_2, D' \rangle$ , where  $\alpha$  is a *response* event. The proof is similar to the third case.
- $(D, D') \in R$ , which means that other concurrent functions change the database state from  $D$  to  $D'$ .

According to the definition of  $\Xi_R$ , we need to prove that  $\langle \sigma_1 \parallel \sigma_2, D' \rangle \Xi_R \langle \Sigma_1 \parallel \Sigma_2, D' \rangle$ .

From  $(D, D') \in R$  and Equation (1), we know

$$\langle \sigma_1, D' \rangle \Xi_R \langle \Sigma_1, D' \rangle. \quad (10)$$

From  $(D, D') \in R$  and Equation (2), we know

$$\langle \sigma_2, D' \rangle \Xi_R \langle \Sigma_2, D' \rangle. \quad (11)$$

From both Equation (10) and Equation (11), we know

$$\langle \sigma_1 \parallel \sigma_2, D' \rangle \Xi_R \langle \Sigma_1 \parallel \Sigma_2, D' \rangle. \quad (12)$$

Therefore, the conclusion Equation (3) is true.  $\square$

Then we prove the following lemma.

**Lemma 2** We use  $A_F$  to denote an automaton running functions in a function set  $F$ . For any function set  $F$  and automaton  $A_F$ , if every function in  $F$  satisfies idempotence simulation, then the following fact holds: from the same initial database state, every time the automaton  $A_{F^*}$  takes one step,  $A_F$  can take  $n$  steps ( $n \geq 0$ ) such that they reach the same database state and produce the same event (if any).

### Proof

The premise is that every function  $f \in F$  satisfies idempotence simulation:

$$\forall f \in F. \quad (\forall D, arg. \langle init(f^*, arg), D \rangle \Xi_R \langle init(f, arg), D \rangle), \quad (13)$$

where  $init(f, arg)$  denotes the initial local state of running the function  $f$  with the argument  $arg$ , and we omit the existential quantifier on  $R$ . The conclusion is that the execution of  $F$  can simulate the execution of  $F^*$ .

We prove the conclusion by classifying every step taken by  $A_{F^*}$  into three cases.

- $A_{F^*}$  creates a function instance to run a function  $f^*$ . Then,  $A_F$  can create an instance with the same identifier to run  $f$  with the same invocation arguments. Both of them do not change the database state and produce the same invocation event. Note that we treat  $f$  and  $f^*$  as the same in invocation events since their only difference is whether to be retried.

- A function instance takes one step. From Equation (13) and Lemma 1, we know that for any functions  $f_1, f_2, \dots \in F$ , function arguments  $arg_1, \dots$ , and shared state  $D$ ,

$$\langle init(f_1^*, arg_1) \parallel \dots, D \rangle \Xi_R \langle init(f_1, arg_1) \parallel \dots, D \rangle. \quad (14)$$

From Equation (14), we know that for any intermediate local states  $(\sigma_{f_1}, \dots)$  and database state  $D'$  during executing functions,

$$\langle \sigma_{f_1^*} \parallel \dots, D' \rangle \Xi_R \langle \sigma_{f_1} \parallel \dots, D' \rangle. \quad (15)$$

According to the definition of  $\Xi_R$ , Equation (15) implies that every time a function instance in  $A_{F^*}$  takes one step, another function instance in  $A_F$  can always take  $k$  ( $0 \leq k$ ) steps to result in the same database state and the same event (if any).

- $A_{F^*}$  retries an instance. Then  $A_F$  takes no steps such that it produces the same database state and no event. The proof is similar to the second case.

Then, the conclusion is true.  $\square$

Finally, we can prove Theorem 1 in the paper.

**Theorem 5** Given a function set  $F$ , if every function  $f \in F$  satisfies idempotence simulation, then  $F$  satisfies idempotence consistency.

### Proof

The premise is that every function  $f \in F$  satisfies idempotence simulation.

$$\forall f \in F. \quad (\forall D, arg. \langle init(f^*, arg), D \rangle \Xi_R \langle init(f, arg), D \rangle). \quad (16)$$

The conclusion is that  $F$  satisfies idempotence consistency (Definition 1).

From Lemma 2 and Equation (16), we imply that the execution of  $F$  can simulate the execution of  $F^*$ . That means every time  $A_{F^*}$  takes one step,  $A_F$  can take  $n$  ( $n \geq 0$ ) steps such that they reach the same database state and produce the same event (if any). Therefore, if some execution of  $A_{F^*}$  can result in the client-observable behavior  $\langle H, D \rangle$ , then there exists another execution of  $A_F$  that can also result in  $\langle H, D \rangle$ . The conclusion is proved.  $\square$

## Appendix B Proof of Failure Reduction

In this section, we prove Theorem 4 in the paper, which proves the second condition in Theorem 3. The first condition in Theorem 3 is intuitive. Thus, we omit its formal proof in this section. We first prove two lemmas and then prove Theorem 4 based on them.

**Lemma 3** If the execution of  $f$  can simulate  $f^1$ , then for any  $n \geq 1$ , the execution of  $f^{n-1}$  can simulate  $f^n$ . The definition of  $f^1$ ,  $f^n$ , and  $f^{n-1}$  are in Section 4.4.

$$\begin{aligned} & (\forall D, arg. \langle \text{init}(f^1, arg), D \rangle \sqsubseteq_R \langle \text{init}(f, arg), D \rangle) \rightarrow \\ & (\forall D, arg, n \geq 1. \langle \text{init}(f^n, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^{n-1}, arg), D \rangle). \end{aligned}$$

### Proof

The premise is

$$\forall D, arg. \langle \text{init}(f^1, arg), D \rangle \sqsubseteq_R \langle \text{init}(f, arg), D \rangle. \quad (17)$$

We want to prove for all  $n \geq 1$ ,

$$\forall D, arg. \langle \text{init}(f^n, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^{n-1}, arg), D \rangle. \quad (18)$$

We prove it by induction on  $n$ .

**Base case:** When  $n = 1$ , Equation (18) is true, because it is equivalent to the premise Equation (17).

**Inductive step:** Suppose Equation (18) is true when  $n = k$  ( $k \geq 1$ ).

$$\forall D, arg. \langle \text{init}(f^k, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^{k-1}, arg), D \rangle. \quad (19)$$

Then when  $n = k + 1$ , we want to prove

$$\forall D, arg. \langle \text{init}(f^{k+1}, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^k, arg), D \rangle. \quad (20)$$

According to the definition of  $\sqsubseteq_R$ , we need to map every single step during the execution of  $f^{k+1}$  to  $s$  ( $s \geq 0$ ) steps during the execution of  $f^k$ . We can construct the mapping in the following way.  $f^{k+1}$  and  $f^k$  are almost the same, except that the platform retries them for different times. Then before the first retry, we map every single step when executing  $f^{k+1}$  to a single step of  $f^k$ . That means  $f^{k+1}$  and  $f^k$  always execute the same statement. This mapping can satisfy the requirements of  $\sqsubseteq_R$ .

When the first retry of  $f^{k+1}$  happens, we ask  $f^k$  to be also retried. Assume the database state immediately before the retry is  $D_1$ . The executions of  $f^{k+1}$  and  $f^k$  from  $D_1$  after the first retry are equivalent to the executions of  $f^k$  and  $f^{k-1}$  from  $D_1$  before the first retry, respectively. This is because after the first retry, the platform will retry  $f^{k+1}$  for  $k$  times and retry  $f^k$  for  $k - 1$  times. From Equation (19), we know that

$$\forall arg. \langle \text{init}(f^k, arg), D_1 \rangle \sqsubseteq_R \langle \text{init}(f^{k-1}, arg), D_1 \rangle. \quad (21)$$

Then there exists a step mapping from every step of  $f^{k+1}$  to steps of  $f^k$  after the first retry, which satisfies the requirements of  $\sqsubseteq_R$ . Therefore, Equation (18) is true for  $n = k + 1$ .

**Conclusion:** By the principle of induction, Equation (18) is true for any  $n \geq 1$ .  $\square$

**Lemma 4** For any  $i, j, k \geq 0$ , if the execution of  $f^k$  can simulate  $f^j$  and the execution of  $f^j$  can simulate  $f^i$ , then the execution of  $f^k$  can simulate  $f^i$ .

$\forall i, j, k.$

$$\begin{aligned} & ((\forall D, arg. \langle \text{init}(f^i, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^j, arg), D \rangle) \wedge \\ & (\forall D, arg. \langle \text{init}(f^j, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^k, arg), D \rangle)) \\ & \rightarrow (\forall D, arg. \langle \text{init}(f^i, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^k, arg), D \rangle). \end{aligned}$$

### Proof

This lemma is similar to the transitivity of forward simulation. The premises include

$$\forall D, arg. \langle \text{init}(f^i, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^j, arg), D \rangle, \quad (22)$$

and

$$\forall D, arg. \langle \text{init}(f^j, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^k, arg), D \rangle. \quad (23)$$

The conclusion is

$$\forall D, arg. \langle \text{init}(f^i, arg), D \rangle \sqsubseteq_R \langle \text{init}(f^k, arg), D \rangle. \quad (24)$$

We use  $\sigma_i$ ,  $\sigma_j$ , and  $\sigma_k$  to denote the local state when executing  $f^i$ ,  $f^j$ , and  $f^k$ , respectively. We will prove the conclusion Equation (24) by *co-induction*. According to the definition of  $\sqsubseteq_R$ , every step taken during executing  $f^i$  belongs to one of the following three cases.

- $\langle \sigma_i, D \rangle \xrightarrow{\alpha} \langle \sigma'_i, D' \rangle$ , where  $\alpha$  is not a *response* event.

According to the definition of  $\sqsubseteq_R$ , we need to prove that there exists  $\sigma'_k$  such that  $\langle \sigma_k, D \rangle \xrightarrow{\alpha} \langle \sigma'_k, D' \rangle$  and  $\langle \sigma'_i, D' \rangle \sqsubseteq_R \langle \sigma'_k, D' \rangle$ .

From Equation (22), we know that there exists  $\sigma'_j$  such that

$$\langle \sigma_j, D \rangle \xrightarrow{\alpha} \langle \sigma'_j, D' \rangle, \quad (25)$$

and

$$\langle \sigma'_i, D' \rangle \sqsubseteq_R \langle \sigma'_j, D' \rangle. \quad (26)$$

From Equation (23) and Equation (25), we know there exists  $\sigma'_k$  such that

$$\langle \sigma_k, D \rangle \xrightarrow{\alpha} \langle \sigma'_k, D' \rangle, \quad (27)$$

and

$$\langle \sigma'_j, D' \rangle \sqsubseteq_R \langle \sigma'_k, D' \rangle. \quad (28)$$

From Equation (26) and Equation (28), we know that

$$\langle \sigma'_i, D' \rangle \sqsubseteq_R \langle \sigma'_k, D' \rangle. \quad (29)$$

- $\langle \sigma_i, D \rangle \xrightarrow{\alpha} \langle \sigma'_i, D' \rangle$ , where  $\alpha$  is a *response* event.

According to the definition of  $\Xi_R$ , we need to prove that there exists  $\sigma'_k$  such that  $\langle \sigma_k, D \rangle \xrightarrow{\alpha} \langle \sigma'_k, D' \rangle$ .

From Equation (22), we know that there exists  $\sigma'_j$  such that

$$\langle \sigma_j, D \rangle \xrightarrow{\alpha} \langle \sigma'_j, D' \rangle. \quad (30)$$

From Equation (23) and Equation (30), we know there exists  $\sigma'_k$  such that

$$\langle \sigma_k, D \rangle \xrightarrow{\alpha} \langle \sigma'_k, D' \rangle. \quad (31)$$

- $\langle D, D' \rangle \in R$ , which means that other concurrent functions change the database state from  $D$  to  $D'$ .

According to the definition of  $\Xi_R$ , we need to prove that  $\langle \sigma_i, D' \rangle \Xi_R \langle \sigma_k, D' \rangle$ .

From Equation (22), we know

$$\langle \sigma_i, D' \rangle \Xi_R \langle \sigma_j, D' \rangle. \quad (32)$$

From Equation (23), we know

$$\langle \sigma_j, D' \rangle \Xi_R \langle \sigma_k, D' \rangle. \quad (33)$$

From Equation (32) and Equation (33), we know

$$\langle \sigma_i, D' \rangle \Xi_R \langle \sigma_k, D' \rangle. \quad (34)$$

Thus, we have proved the conclusion Equation (24).  $\square$

Finally, we can prove Theorem 4 in the paper based on the above two lemmas.

**Theorem 6** Given a function  $f$ , if each execution with one retry under concurrency has a corresponding retry-free execution that can simulate it, then each execution with arbitrary times of retries also has a corresponding retry-free execution that can simulate it.

$$\begin{aligned} & (\forall D, arg. \langle \text{init}(f^1, arg), D \rangle \Xi_R \langle \text{init}(f, arg), D \rangle) \rightarrow \\ & (\forall D, arg, n \geq 1. \langle \text{init}(f^n, arg), D \rangle \Xi_R \langle \text{init}(f, arg), D \rangle). \end{aligned}$$

### Proof

The premise is

$$\forall D, arg. \langle \text{init}(f^1, arg), D \rangle \Xi_R \langle \text{init}(f, arg), D \rangle. \quad (35)$$

The conclusion is

$$\forall D, arg, n \geq 1. \langle \text{init}(f^n, arg), D \rangle \Xi_R \langle \text{init}(f, arg), D \rangle. \quad (36)$$

From Lemma 3 and the premise Equation (35), we know

$$\forall D, arg, n \geq 1. \langle \text{init}(f^n, arg), D \rangle \Xi_R \langle \text{init}(f^{n-1}, arg), D \rangle. \quad (37)$$

From Lemma 4 and Equation (37), we know

$$\forall D, arg, n \geq 1. \langle \text{init}(f^n, arg), D \rangle \Xi_R \langle \text{init}(f^0, arg), D \rangle. \quad (38)$$

Since  $f^0$  is equivalent to  $f$ , Equation (38) is equivalent to the conclusion Equation (36). The conclusion is true.  $\square$

```

1 int f(input)
2 {
3   output = f1(input);
4   return f2(output);
5 }

```

**Figure 12:** A function composed of two sub-functions called  $f_1$  and  $f_2$ .

## Appendix C Proof of Theorem 2

This section proves Theorem 2 in Section 4.3 of the paper. We first define and prove sequential compositionality of idempotence simulation.

**Definition 2** Given a function  $f$  and the rely condition  $R$ ,  $f$  satisfies strong idempotence simulation means that: 1)  $f$  satisfies idempotence simulation under  $R$ ; and 2) after the platform successfully executes  $f$  without retries for one time, retrying  $f$  again will not modify the shared state.

Particularly, the second condition is equivalent to the third requirement in Theorem 2 of paper: it will not affect the shared state on retry once it has been successfully executed.

**Lemma 5** Given any two functions  $f_1$  and  $f_2$ , if  $f_1$  satisfies strong idempotence simulation,  $f_2$  satisfies idempotence simulation, and the input of  $f_2$  remains unchanged on retry, then the function  $f$  in Figure 12 composed of  $f_1$  and  $f_2$  also satisfies idempotence simulation.

**Proof** According to Theorem 3, we prove the simulation relation between  $f^1$  and  $f$ . Note that  $f^1$  is defined in Section 4.4, which is different from  $f_1$ . Then we classify the location where retry occurs during the execution of  $f^1$  into three cases and prove that under all these cases, executing  $f$  without retries could exhibit all possible client-observable behaviors produced by executing  $f^1$ . Then we can prove that  $f$  satisfies idempotence simulation.

- Retry happens during the execution of  $f_1$ . Then the execution of  $f^1$  consists of three main parts:
  - (P1) the execution of  $f_1$  before retry;
  - (P2) the normal execution of  $f_1$  after retry;
  - (P3) the normal execution of  $f_2$ .
 Since  $f_1$  satisfies idempotence simulation, a normal execution of  $f_1$  without retries could exhibit all client-observable behaviors of (P1) and (P2). In this case, a normal execution of  $f$  without retries can simulate the execution of  $f^1$ .
- Retry happens between  $f_1$  and  $f_2$ . This execution exhibits the same client-observable behavior as another execution where retry happens immediately before the “return” statement in  $f_1$ . The proof is the same as the first case.
- Retry happens during the execution of  $f_2$ . Then the execution of  $f^1$  consists of four main parts:
  - (P1) the first normal execution of  $f_1$  without retries;

```

1 int f(int input)
2 {
3     int output1 = f1(input);
4     int output2 = f2(output1);
5     ...;
6     return fn(outputn);
7 }

```

**Figure 13:** A function composed of  $n$  sub-functions called  $f_1, \dots,$  and  $f_n$ .

- (P2) the execution of  $f_2$  before retry;
- (P3) the second normal execution of  $f_1$  without retries;
- (P4) the normal execution of  $f_2$  without retries.

Since  $f_1$  satisfies strong idempotence simulation, the second normal execution of  $f_1$  will not modify the shared state and return the same value as the input for the  $f_2$ . Therefore, this kind of execution of  $f^1$  exhibits the same client-observable behavior as the execution composed of (P1), (P2), and (P4). Since  $f_2$  satisfies idempotence simulation, there exists another normal execution of  $f_2$  that exhibits the same client-observable behavior as the execution of (P2) and (P4). Thus, the execution of  $f^1$  described in this case exhibits the same client-observable behavior as another normal execution of  $f$  without retries.

In conclusion, for any execution of  $f^1$ , there always exists a normal execution of  $f$  that exhibits the same client-observable behavior under concurrency. According to Theorem 4,  $f$  satisfies idempotence simulation.  $\square$

Then we extend sequential compositionality to a function composed of arbitrary number of code fragments.

**Lemma 6** For any positive integer  $n \geq 2$  and any function  $f$  in the form of Figure 13, if each  $f_i$  ( $1 \leq i < n$ ) satisfies strong idempotence simulation,  $f_n$  satisfies idempotence simulation, and the input of each  $f_i$  remains unchanged on retry, then  $f$  satisfies idempotence simulation.

**Proof** We prove it by induction on  $n$  ( $n \geq 2$ ). The premise is Lemma 5.

**Base case.** When  $n = 2$ , the conclusion is true, because it is equivalent to the premise.

**Inductive step.** Assume the conclusion is true when  $n = k$ . We prove that the conclusion is also true when  $n = (k + 1)$ . We can treat the code fragment containing the first  $k$  sub-functions as a function  $g$ . Since the conclusion is true when  $n = k$ , the function  $g$  satisfies idempotence simulation.  $f$  consists of two sub-functions called  $g$  and  $f_{k+1}$ , both of which satisfy idempotence simulation. From Lemma 5, we can imply that  $f$  satisfies idempotence simulation.

**Conclusion.** By the principle of induction,  $f$  satisfies idempotence simulation for any  $n \geq 2$ .  $\square$

Based on Lemma 6, we prove that our method of addressing unbounded loops with write operations is sound, which is

```

1 void checkCoupons(coupons, time) {
2     // C1
3     if(coupons.size == 0)
4         return;
5     // L
6     for(int j = 0; j < coupons.size(); j++) {
7         coupon := get("Coupon", coupons[j].couponId);
8         if(isExpired(coupon.date, time)) {
9             coupon.expired := true;
10            put("Coupon", coupons[j].couponId, coupon);
11        }
12    }
13    // C2
14    return;
15 }

```

**Figure 14:** An example of unbounded loop with write operations.

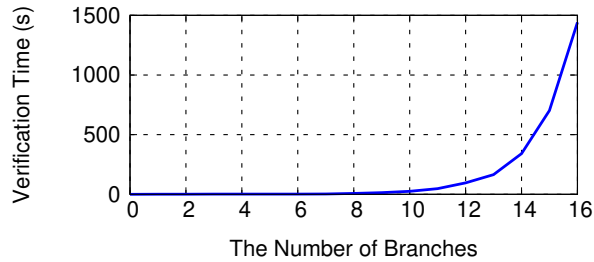
Theorem 2 in the paper. For convenience, we represent a function with an unbounded loop as  $\{C_1; L; C_2\}$ , where  $L$  is the unbounded loop,  $C_1$  is all code preceding  $L$ , and  $C_2$  denotes all code following the loop.  $B_L$  is the loop body of  $L$ .

**Theorem 7** Given a function  $f$  with the unbounded loop in case 2,  $f$  satisfies idempotence simulation if the number of iterations of the loop  $L$  remains unchanged on retry, and  $C_1$ ,  $C_2$  and  $B_L$  can satisfy the following requirements: 1) They all satisfy idempotence simulation; 2) Their inputs do not change on retry; 3) They will not affect the shared state on retry once the function has successfully executed them.

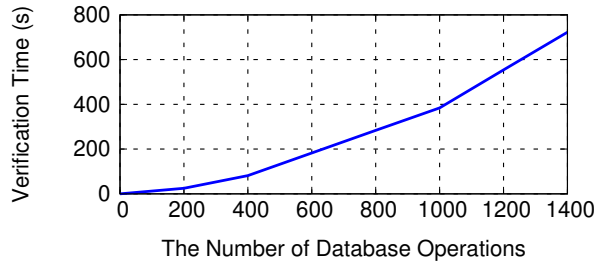
**Proof** Since the number of loop iterations is the same on retry, the execution of  $f$  comprises the execution of  $C_1$ , the execution of an unbounded number of  $B_L$ , and the execution of  $C_2$ . We can prove that  $f$  satisfies idempotence simulation based on Lemma 6. Although the number of loop iterations is unbounded, the loop body executed by each iteration is the same. Therefore, we just need to prove that  $C_1$ ,  $B_L$ , and  $C_2$  satisfy the requirements in Lemma 6. The requirements have been ensured by the premise of Theorem 7. Therefore, the conclusion is true and  $f$  satisfies idempotence simulation.  $\square$

We use an example to show how to use this theorem to perform the verification. In Figure 14, the `checkCoupons` function uses an unbounded loop to check whether coupons have expired. Obviously,  $C_1$  and  $C_2$  in `checkCoupons` satisfy all requirements in Theorem 2. The number of the loop iterations is the size of `coupons` which will be the same on retry. Thus, to prove the idempotence simulation of `checkCoupons`, we only need to focus on the loop body  $B_L$  (line 7-11). First, Flux can prove that  $B_L$  satisfies the idempotence simulation; Then, it uses static analysis to find that  $B_L$ 's input is `coupons` that keeps consistent on retry. Third, once it successfully updates `expire` to be `true` for a specific `coupon`, the value will remain unchanged as `checkCoupons` always updates it to be `true` on retry. According to the theorem, we have that `checkCoupons` satisfies the idempotence simulation.

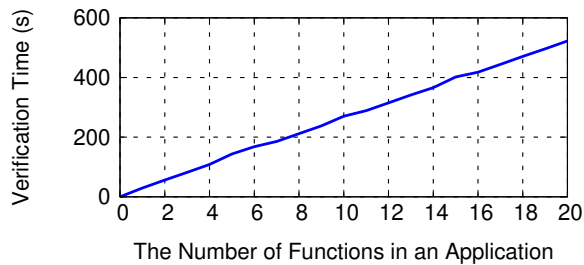




(a) The verification time of a single function with ten database operations and different numbers of branches.



(b) The verification time of a single function with different numbers of database operations. The function has one execution path. Lines of code also increase linearly when the number of database operations increases linearly.



(c) The verification time of an application with different numbers of functions.

**Figure 15:** The verification time for different numbers of branch statements, database operations, and functions.

## Appendix D Scalability of the Verifier

We have created micro-benchmarks to evaluate the scalability of the verifier. Figure 15a shows that when the number of branches in a single function increases, the verification time increases exponentially, as the number of traces also increases exponentially. Besides, Figure 15b shows that when the number of database operations in a single function increases, the verification time increases linearly. This is because the number of generated traces increases linearly. Note that because these micro-benchmarks mainly contain database operations, LoC also increases linearly when the number of database operations increases. Thus, Figure 15b also demonstrates that when the LoC of a single function increases, the verification time increases linearly. Additionally, we evaluate the verification time for an application with different numbers

of functions. Each function has one execution path and two hundred database operations. Figure 15c shows that the verification time increases linearly when the number of functions in an application increases linearly. Because the number of traces increases linearly.



# Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems

Travis Hance<sup>†</sup>, Yi Zhou<sup>†</sup>, Andrea Lattuada<sup>‡,\*</sup>, Reto Achermann<sup>\*</sup>, Alex Conway<sup>‡</sup>,  
Ryan Stutsman<sup>‡,⊕</sup>, Gerd Zellweger<sup>‡</sup>, Chris Hawblitzel<sup>⊖</sup>, Jon Howell<sup>‡</sup>, Bryan Parno<sup>†</sup>

<sup>†</sup> *Carnegie Mellon University*; <sup>\*</sup> *University of British Columbia*;  
<sup>‡</sup> *VMware Research*; <sup>⊕</sup> *University of Utah*; <sup>⊖</sup> *Microsoft Research*

## Abstract

We present IronSync, an automated verification framework for concurrent code with shared memory. IronSync scales to complex systems by splitting system-wide proofs into isolated concerns such that each can be substantially automated. As a starting point, IronSync’s ownership type system allows a developer to straightforwardly prove both data safety *and* the logical correctness of thread-local operations. IronSync then introduces the concept of a *Localized Transition System*, which connects the correctness of local actions to the correctness of the entire system. We demonstrate IronSync by verifying two state-of-the-art concurrent systems comprising thousands of lines: a library for black-box replication on NUMA architectures, and a highly concurrent page cache.

## 1 Introduction

Despite the importance of concurrent software, it is famously difficult to write correctly. The correctness of any one thread can, in principle, depend on changes from any other thread; developers often struggle to consider all possible thread interleavings. This reasoning becomes more difficult in aggressively optimized systems that use custom synchronization tools beyond standard abstractions like locks. Such systems often appear *anti-modular* in that they entangle synchronization logic with application logic. For example, a concurrent page cache (§5.2) might use a bit both for the synchronization purpose of read-locking a section of memory and for the logical purpose of indicating that an IO write is in progress.

In theory, formal software verification can produce provably correct code, but existing techniques have struggled to reach production-scale shared-memory systems (§9). Some work [19, 20] on verifying concurrent systems carefully focuses on networking and asynchronous disk IO, but avoids *shared-memory* concurrency. Other work [11, 12, 17, 36, 46] does tackle shared memory, but it relies on techniques which require considerable sophistication and manual effort from the developer. Still other work [38] offers simpler tools and greater automation, but the automation does not yet scale, requiring hours of CPU time for tens of lines of code (§7.1).

In contrast, IronSync enables verification of production-scale shared-memory concurrent systems, including those with custom synchronization protocols. Such verification comes with many reasoning challenges, and IronSync suc-

ceeds by carefully partitioning the system-level proof so that a developer can ergonomically tackle each challenge by formalizing her existing intuitions, supported at each stage by powerful automation.

At the lowest level, an IronSync developer uses an ownership type system to prove the data safety of her implementation. Oversimplifying, in an ownership type system, an *owned* value must be held (or referenced) by exactly one variable. A fast, deterministic type checker enforces this property, which IronSync, like Rust [28, 39] and other languages, in turn uses to enforce data safety, i.e., basic memory safety plus freedom from conflicting reads and writes. Our experience, and that of the Rust community, suggests using an ownership type system for such reasoning is relatively intuitive.

Going further, IronSync shows how the developer can additionally use ownership types to reason about the *logical correctness* of a thread’s local actions on data it owns. Intuitively, in any segment of code where a thread operates only on owned data, that data cannot affect or be affected by other threads. Hence, IronSync can reason about such code using *sequential* reasoning techniques and tools honed over decades of research and development. Reasoning sequentially is more intuitive for the developer as well.

Finally, the IronSync developer must connect the locally correct thread actions to global, *system-wide* correctness. To support this step, IronSync introduces the *Localized Transition System* (LTS), which abstracts each thread’s actions and the state they act upon, formalizing the intuition that in efficient concurrent programs, each thread acts only on a small fragment of the program’s global state. Using the techniques above, the developer locally proves that the LTS is a sound abstraction of their implementation. Finally, IronSync soundly abstracts the LTS into a simplified program representing the whole system. This new program (with threads, locks, and other implementation details abstracted away) is far simpler to reason about, and it connects naturally to previous automated techniques for, e.g., reasoning about asynchronous IO or distributed nodes [19, 20]. Since each step on this path is proven sound, proofs at this level also apply to the implementation.

To support the advanced read-sharing patterns found in production systems, IronSync also reuses the machinery above (with a small twist) to factor out reasoning about complex synchronization primitives, so that the developer can think about them separately from the core application logic, even if

\*Work done while at ETH Zurich

the implementation deeply entangles the two concerns.

The developer writes their implementation and performs all of the reasoning above in an extended version of the Dafny language [33] augmented with the trusted axioms and memory primitives in IronSync’s framework.

**Evaluation.** We introduce IronSync via a series of increasingly complex examples, culminating in two production-level case studies (§5). To illustrate IronSync at scale, we verify a Node Replication (NR) library that creates a linearizable NUMA-aware concurrent data structure from a black-box sequential one [9]. To show IronSync working both at scale and with prior work on crash safety [19], we verify Splinter-Cache [14], a production-scale disk-backed in-memory page cache created for use in commercial products [49].

Each case study’s performance matches its unverified production-level counterpart (§7.2), driving workloads of 5M updates/sec. with 192 threads (NR) or 3M ops/sec. (SplinterDB with verified SplinterCache), demonstrating IronSync has not impeded optimization through limited expressiveness or excessive proof burden. We also uncovered severe bugs (§7.3) in the unverified implementations.

**Limitations.** As in any verification system, the correctness of a verified IronSync program depends on the correctness of its spec, and the verification tool (Dafny [33]). Our encoding of the IronSync framework in Dafny is also trusted (although application-specific definitions are not). IronSync does not verify liveness, termination, or deadlock-freedom. We focus on data safety and functional correctness; i.e., *if* an operation returns a result, it is correct according to its spec. IronSync verifies programs against a high-level memory consistency model that distinguishes data-race-free memory from racy, sequentially consistent atomic memory. This compiles to efficient assembly on modern hardware, but it cannot exploit every optimization afforded by relaxed memory (§4.2).

**Contributions.** In summary, this paper:

1. Factors the proof of a production-scale concurrent system into intuitive reasoning steps that can each be supported with powerful automation.
2. Illustrates how the application of an ownership type system enables scalable, automated concurrent reasoning about both data safety *and* logical correctness.
3. Introduces Localized Transition Systems, which soundly connect *local* reasoning about a thread’s actions to *global* reasoning about the correctness of the full system.
4. Enables developers to verify complex, application-specific read-sharing synchronization tools in isolation from the program’s main application logic.
5. Demonstrates, via case studies, that IronSync effectively reasons about practical, complex, high-performance concurrent systems.

## 2 The Potential Pitfalls of Parallelism

To highlight the challenges of writing and reasoning about concurrent code, we begin with a simple bank application.

```
if accounts[A] ≥ amt {
  accounts[A] = accounts[A] - amt
  accounts[B] = accounts[B] + amt
}
```

Figure 1: Buggy code violating data safety

```
lock(accounts[A]);
sufficient_balance := accounts[A] ≥ amt;
unlock(accounts[A]);

if sufficient_balance {
  lock(accounts[B]);
  accounts[B] = accounts[B] + amt
  unlock(accounts[B]);

  lock(accounts[A]);
  accounts[A] = accounts[A] - amt
  unlock(accounts[A]);
}
```

Figure 2: Buggy code violating logical correctness

```
lock(accounts[A]);
lock(accounts[B]);
if accounts[A] ≥ amt {
  accounts[A] = accounts[A] - amt
  accounts[B] = accounts[B] + amt
}
unlock(accounts[A]);
unlock(accounts[B]);
```

Figure 3: Correct code (assuming that  $A < B$ , which is necessary to avoid deadlock, although proving deadlock-freedom is out-of-scope for IronSync)

This multi-threaded application maintains a list of accounts, each with an account ID and a balance, supporting one operation,  $transfer(A, B, d)$ , which moves  $d$  dollars from account  $A$  to account  $B$ .

Here, there are a handful of mistakes an inexperienced developer might make. First, they might write code as in Figure 1, which would be correct in a sequential program but critically flawed in a concurrent one, where, e.g., two different threads might both write to `accounts[A]`, one overwriting the other. In fact, it might be difficult to even fully characterize the buggy behavior on realistic hardware with weaker memory models [1, 2, 45], and in some programming languages, data races may even be undefined behavior [5, 52].

A naive fix would be to protect the code in Figure 1 with a global lock. Such a fix would be correct but embarrassingly inefficient. For efficiency, the developer might employ a finer-grained concurrency strategy by creating a lock for each account, reasoning that most transfers affect disjoint accounts.

Figure 2 uses a discipline of locking an account before reading/writing it, eliminating the *data-safety* problems above, since no other thread can simultaneously read/write the account information. Even with this fix, however, the program is still not *logically correct*: two different transfer operations might each check that account  $A$  has sufficient funds and then move forward even when  $A$  lacks the funds to complete both, leaving  $A$  with a negative balance.

Given enough debugging (or prior experience), the developer might eventually produce the code in Figure 3, which

holds locks on both accounts as it makes the transfer. How would the developer convince herself that she has finally produced a correct implementation?

She might first reason that the program is *data safe* because it holds the corresponding account lock whenever it accesses shared memory. Data safety rules out blatant data corruptions, making it feasible to reason about logical correctness.

Given data safety, she might then informally reason that holding both locks simultaneously gives the thread exclusive access to the portion of the state (accounts *A* and *B*) needed to correctly perform the transfer. Although other concurrent threads might simultaneously modify *other* program state, all such modifications are irrelevant to the transfer. Conversely, any changes the thread makes to accounts *A* and *B* must be irrelevant to any concurrent transfers, since those transfers must involve other accounts. Hence, she can *reason locally* about the correctness of the transfer implementation. Furthermore, from the perspective of the other threads, the transfer appears to occur atomically.

With IronSync, the developer formalizes her intuitions about data safety and logical correctness in a machine-checked way.

### 3 The Core IronSync Methodology

IronSync utilizes a variety of techniques, with the philosophy of using the right tool for each job. We introduce the techniques through a series of increasingly complex examples.<sup>1</sup> Here we focus on the toy banking application from §2 to illustrate IronSync’s core ideas: (1) the use of an ownership-based type system, (2) the abstraction of threads via a *localized transition system*, and (3) the way we soundly compose a *localized transition system* into a *global state machine* that can soundly reason about global properties of the full concurrent program.

In §4, we introduce increasingly sophisticated features and examples, building up to our production-level case studies in §5. We defer the formalism underpinning IronSync to §6.

#### 3.1 Achieving Data Safety in IronSync

IronSync mechanically enforces data-safety via an ownership type system. Such type systems are effective at enforcing data-safety both in unverified programming (e.g., in Rust [29]) and in verified *sequential* programming; e.g., in *Linear Dafny* [35], a version of the Dafny verification language [33] augmented with an ownership (or linear [51]) type system inspired by Rust’s. In IronSync, we extend Linear Dafny with tools for logical correctness in *concurrent* settings (3.2). First, though, we overview ownership types, explain how they enforce data safety, and how this aids in verification.

In Linear Dafny, an *owned* value must be held (or referenced) by exactly one variable. Any attempts to duplicate or drop the value are rejected by the type checker. In IronSync, this ensures that data is *uniquely owned*, and hence a

<sup>1</sup>All examples are fully verified and available in our open-source release.

thread can read or write to data it owns without interference from other threads. Owned values can, however, be stored in shared (read-only) variables; Dafny’s type checker ensures that the scope of such shared variables is contained within the scope of the originating owned variable, and that the owned variable is not modified until the shared variables expire.

```
method M(owned w: int, b: bool) returns (owned z: int)
  owned var x := w; // okay: consumes w
  owned var y := w; // error: w was already consumed
  if b {
    shared var s := x; // okay: shares x read-only
    x := x + 1; // error: borrowed value still live in s
  }
  x := x + 1; // okay: shared variable s has expired
  z := x; // okay: consumes x
```

Figure 4: Ownership in action

IronSync uses the ownership type system to ensure that code cannot read or write shared memory after it gives up permission to access it. Figure 5 shows an example of this approach in a portion of the API for an exclusive lock (which in turn is implemented and proven correct using lower-level IronSync primitives – see §4.2). The API uses owned and shared variables, as introduced above, and generic types (indicated by the type parameter <T>) as in Java or C#. Notice that the `Lock` itself is passed `shared`; hence, it can be referenced simultaneously from multiple threads, as expected of a lock. The caller obtains an owned `guard` value when it acquires the lock (`AcquireExcl`). The guard object, named after objects like C++’s `std::lock_guard` or Rust’s `RwLockWriteGuard`, is an object that exists for the duration the lock is held. Furthermore—and similarly to `RwLockWriteGuard`—the guard object provides a type-safe means to access the data being protected by the lock. That is, after the user relinquishes the guard on release (`ReleaseExcl`), the type system will reject any further attempts to use the guard to access the lock’s data.

```
method AcquireExcl<T>(shared m: Lock<T>)
  returns (owned guard: ExclGuard<T>)
method ReleaseExcl<T>(owned guard: ExclGuard<T>)
```

Figure 5: Example lock specification.

With these tools, an IronSync developer can rule out data-safety issues, because shared memory can only be accessed while holding the corresponding permission (e.g., the `ExclGuard`) from a lock (or a fancier primitive – see §4.2). Correct management of the permission is enforced by the ownership type system.

Using an ownership type system to enforce data safety is not unique to verification; e.g., this is a crucial feature of Rust. Novel to IronSync, however, is its further use of its ownership system to help the developer verify logical correctness.

#### 3.2 Local Logical Correctness

IronSync uses ownership to simplify and automate reasoning about the correctness of a thread’s actions on data it owns.



### 3.2.1 Ownership Simplifies Concurrent Correctness

With IronSync, we observe that a type-enforced ownership discipline dramatically simplifies reasoning about *correctness* in concurrent settings. To illustrate, consider this short Dafny program, using its traditional heap model (i.e., without ownership types):

```
method M(data: Data)
  modifies data
  requires data.x == 2
  ensures data.x == 3
{
  data.x := data.x + 1;
}
```

The Dafny verifier can easily prove that if the precondition in the **requires** clause holds, then the postcondition in the **ensures** clause will always hold. However, if this method were part of a concurrent program, Dafny’s standard sequential reasoning would not be sound, since another thread could change the heap-allocated data at any time.

With ownership types, however, we can rewrite it as:

```
method M(owned data: Data)
  requires data.x == 2
  ensures data.x == 3
{
  data.x := data.x + 1;
}
```

Because the type system ensures that the data value is uniquely owned, the verifier can once again make the assumption that no other thread is concurrently modifying data. Hence, the verifier can soundly use the same algorithms as before to easily verify this method.

In short, because IronSync mediates all access to shared resources via Linear Dafny’s ownership type system, we can verify the logical correctness of concurrent code operating locally on owned values by using algorithms and tools that have been honed for decades on sequential verification. In our experience, this brings a substantial boost in proof automation.

### 3.2.2 Maintaining Local Correctness with Invariants

However, even in a program that obeys an ownership discipline, one thread seldom owns a piece of data indefinitely; instead, threads hand off ownership of shared data via synchronization primitives like locks. Hence, the developer needs a mechanism to reason about what value(s) shared data may hold when a thread acquires ownership of that data.

In one such mechanism, IronSync allows the developer to reason about locked data by associating each lock with a *lock invariant*, i.e., a property of the data protected by the lock (Figure 6). The thread can *assume* the property holds when it acquires the lock, and in exchange, it must *prove* the property holds when it releases the lock. This in turn means that the next thread to take the lock may also “take” the assumption.

Again, we see the utility of ownership types: the verifier can soundly assume that the value of `guard.v` is not being modified by other threads while the lock is held, since the

```
function is_even(x: int) { x % 2 == 0 }

// Create a new lock with an invariant that its value
// is always even. Supply a compliant initial value (2).
shared var m := NewLock(2, is_even);
owned var guard := AcquireExcl(m);
assert guard.v % 2 == 0; // Passes
guard.v := guard.v + 1;
// ReleaseExcl(guard); // error: violates invariant `is_even`
guard.v := guard.v + 1;
ReleaseExcl(guard); // okay: satisfies invariant `is_even`
// guard.v := guard.v + 1; // error: guard already consumed
// ReleaseExcl(guard); // error: guard already consumed
```

Figure 6: Example of a lock invariant. Any commented line, if uncommented, would give the resulting error. Note that `ReleaseExcl` consumes the **owned** guard object, so it cannot be used later.

thread holding the lock uniquely owns the guard. Hence, we can continue to use the efficient sequential verification techniques from §3.2.1 to prove the correctness of a thread’s actions on data it obtains from other threads.

### 3.3 From Local to Global Logical Correctness

We have shown IronSync’s use of ownership to establish data safety and the logical correctness of a thread’s local actions on data it owns. The final step is to explain how threads cooperate to achieve a global (program-wide) logical goal.

This final step would be trivial in a program that protects all of its state with a global lock. In that case, proofs of local correctness would suffice for global correctness, since we would simply specify the program’s expected behavior via an invariant on the global lock. Such an approach would be correct but embarrassingly inefficient.

For better efficiency, the developer might employ a finer-grained concurrency strategy using many local locks. At this point, proving a global property directly becomes difficult since no thread has a global view of the system.

For such systems, IronSync provides the developer with tools to build up to global logical correctness in stages. First, the developer creates a simplified abstract model of the threads’ local actions (§3.3.1). Second, the developer uses the techniques from §3.2 to *locally* prove that each thread’s implementation can be soundly abstracted by the model (§3.3.2). Finally, IronSync reassembles the model’s local actions into a single global abstraction of the program (§3.3.3). At this level, the developer can reason about global properties of the system, without the complexity of low-level details like thread interleaving, memory management, or even locking strategy. Because each step above is proven sound, the global correctness properties hold for the implementation as well.

#### 3.3.1 Abstracting Local Actions

As a first step towards proving global correctness, an IronSync developer proves that their implementation corresponds to a simpler program, where threads, locks, and other implementation details are entirely abstracted away. Reasoning about the correctness of the new program is much simpler.

Returning to our bank, we would like the abstract program to operate over a simple state representing *all* of the accounts:

```
State: {accounts: map<AccountId, Balance>}
```

The challenge is to connect the application’s concrete state (e.g., as stored in array) to the abstract state above.

As discussed earlier, once the implementation commits to a fine-grained, per-account locking strategy (as in Figure 3), it becomes difficult for an individual thread to reason about the global concrete state. After all, any given thread holds at most two locks at a time, and hence it cannot authoritatively reason about the state of the rest of the accounts.

Hence, IronSync introduces the concept of a *Localized Transition System (LTS)* (formally defined in §6.1), which breaks the abstract program’s state into *shards* that match the “granularity” of the concrete implementation. The LTS then defines transitions that apply *locally* to only a subset of the shards. These transitions capture the work a thread performs on its local view of the state. We can later (§3.3.3) reassemble these local views into a global view.

Figure 7 shows the LTS definition for our bank example. The LTS defines a shard to be the information for a single account, matching the granularity of the locking scheme in Figure 3. The localized transition function `transfer` says that a thread that holds two shards (the “pre-state shards”), one for *A* and one for *B*, where *A*’s shard holds at least *amt* dollars, can exchange those shards for a new pair of shards (the “post-state shards”) with updated balances. This definition directly captures the intuition that a transfer only affects (and is affected by) the state of the two accounts involved. All other accounts (shards) are irrelevant.

Notice the abstraction the LTS provides: the update to the shards occurs atomically without any explicit mention of particular synchronization primitives. For complex systems, this simplification makes the application vastly easier to analyze.

In designing their LTS, a developer will typically choose a “granularity” for their shards and actions that matches the granularity of the implementation’s concurrency strategy. As we discuss below, this makes it feasible to use the local correctness techniques from §3.2 to tie the implementation to the LTS. Choosing a coarser granularity would complicate the proof of this connection, while choosing a finer granularity would introduce unnecessary complexity into proofs about the global system (§3.3.3).

In practice, this means that different programs will use different LTS designs. A program with a modest concurrency strategy can afford to use coarse-grained shards, doing most of the proof work “locally” using techniques from §3.2. A program with an aggressive fine-grained concurrency strategy will use finer-grained shards, and thus put more work into spanning the gap from the LTS to the global system.

### 3.3.2 Tying the Concrete Implementation to the LTS

To make use of the abstraction provided by the LTS, we must soundly (i.e., in a machine-checked way) establish that

```
Shard: {id: AccountId, balance: Balance}

localized transition transfer(A, B, amt):
  for some (bal_1, bal_2) where bal_1 ≥ amt,
  pre-state shards:
    {id: A, balance: bal_1}
    {id: B, balance: bal_2}
  post-state shards:
    {id: A, balance: bal_1 - amt},
    {id: B, balance: bal_2 + amt}
```

Figure 7: Bank LTS

the implementation’s behavior matches that of the LTS; thus properties proved about the LTS will meaningfully apply to the real implementation.

A key idea in IronSync is that we tie the implementation to the LTS by explicitly manifesting and manipulating the state shards of the LTS *abstraction* in the *implementation* code. The code then uses the local correctness techniques from §3.2 to prove that its manipulation of its concrete state correctly reflects LTS-defined actions on the corresponding shards.

In more detail, the implementation holds LTS shards in owned *ghost* variables. Ghost variables act like normal variables, but they serve only as “proof constructs” and are absent from the compiled executable. Making the shards owned ensures that they cannot be duplicated, preventing the implementation from holding on to two potentially contradictory shards (e.g., one that claims account *A* holds *v* dollars and another that claims it holds *v + x* dollars). Indeed, unique ownership prevents those shards from existing anywhere in the system, even spread across different threads.

In practice, an IronSync developer will typically embed the ghost shards into their implementation and tie the ghost state to physical state via invariants. The top of Figure 8 illustrates this idea for our bank example. The implementation stores each account’s concrete balance in an `Entry` datatype (similar to a `struct`) that also holds a ghost owned shard defined by the LTS. The account entries live in a sequence, each protected with a lock with an invariant that the ghost state in the shard matches the physical state of the implementation.

Looking at this program, a reader might understandably wonder what is accomplished by redundantly “doubling up” the state into a physical account balance and a ghost account balance. The key is that by doing so, we establish the formal correspondence between the concrete implementation state and the abstract state of the LTS.

The final step is to connect the implementation’s *actions* to the transitions in the LTS. To do so, IronSync provides a trusted, axiomatic API for the ghost shards, with API calls that update the shards by performing valid transitions of the verified LTS. Each call *consumes* the old owned shards, and *produces* new owned shards. As shown in Figure 8, this means that during a transfer, the developer can update the physical state of the account balances and then atomically exchange (via `LTS_transition`) the old shards for a new pair representing the LTS state after performing the abstract transfer transition. These new shards match the concrete

```

datatype Entry = Entry(bal: int, ghost owned shard: Shard)

shared var accts : seq<Lock<Entry>> where
  ∀i, accts[i] has lock invariant: (entry: Entry)
    ⇒ entry.shard == Shard({id: i, balance: entry.bal})

method DoTransfer(A: AccountId, B: AccountId, amt: Dollars)
  // Acquire locks on both accounts.
  owned var guardA := AcquireExcl(accts[A]); lock A
  owned var guardB := AcquireExcl(accts[B]); lock B

  // These follow from the lock invariant.
  assert guardA.v.shard.bal == guardA.v.bal;
  assert guardB.v.shard.bal == guardB.v.bal;

  // Physically move `amt` from one account to the other.
  guardA.v.bal -= amt; debit A
  guardB.v.bal += amt; credit B

  // Invariant is temporarily broken.
  assert guardA.v.shard.bal != guardA.v.bal;

  // Perform the transfer transition of the LTS
  // as a ghost operation.
  guardA.v.shard, guardB.v.shard := ghost transition
    LTS_transition("transfer", guardA.v.shard, guardB.v.shard, amt);

  // Lock invariants have been restored.
  assert guardA.v.shard.bal == guardA.v.bal;
  assert guardB.v.shard.bal == guardB.v.bal;

  // We can now release the locks.
  ReleaseExcl(guardA); unlock A
  ReleaseExcl(guardB); unlock B

```

Figure 8: An implementation of our bank example. Figure 9 illustrates one possible execution.

state, satisfying the corresponding lock invariants and hence allowing the locks to be released.

Hence, we can soundly reason about the implementation’s concrete actions on concrete state using the LTS’s abstract transitions on its abstract shards. IronSync’s trusted API allows the programmer to make this connection locally in the implementation code by showing that a sequence of physical steps are consistent with the “large” atomic steps of the LTS.

We illustrate this process in Figure 9a, which shows one invocation of the DoTransfer method from Figure 8. The illustration depicts the relationship between the ghost shards of the LTS (dashed blue boxes) and the physical values stored in memory. Time runs along the x-axis; each vertical gradation represents a fine-grained period, such as a single instruction.

Initially, Thread 2 holds no locks. It receives a client request to transfer \$7 from *A* to *B*. The developer knows that the relevant LTS transition requires atomically interacting with the *A* and *B* shards, so the thread’s first step is to acquire lock *A*. Lock acquisition brings into Thread 2’s scope both permission to observe the physical value of *A* (via a pointer; the physical value does not move from the heap, of course) as well as the ghost shard for *A*.

Later, Thread 2 likewise acquires the physical *B* state and its ghost shard. The ghost LTS transition requires that shard *A* has a value greater than the \$7 transfer. Thread 2 confirms this by checking the physical value of *A*, which it knows (from the lock invariant) matches the ghost shard for *A*.

Then Thread 2 debits \$7 from the physical value of *A*. Note that the ghost shard has not changed; no LTS transition allows debiting *A* all by itself. The lock invariant is temporarily false, which is fine, since Thread 2 still holds the lock. Next, Thread 2 credits \$7 to the physical value of *B*.

Thread 2 cannot release the locks until it restores their invariants. Hence, it invokes transfer(*A*,*B*,7), the ghost LTS transition from Figure 7, which consumes the shards *A* : 9, *B* : 1. As a ghost transition, this happens instantaneously. The transition yields (by postcondition) the new shards *A* : 2, *B* : 8, which the thread proves match the corresponding physical values. Having restored the lock invariants, the thread completes its work by releasing its locks, one at a time.

For simplicity, we do not illustrate any activity on Thread 1. However, observe that it could, with any interleaving, acquire noncontending locks and interact with their associated state.

### 3.3.3 Global Logical Correctness With the GSM

To reason about the logical correctness of the entire multi-threaded program, we reassemble the shards of the LTS into a representation of the program’s global state, and similarly, we translate the local transitions of the LTS into global transitions over the global state. We call the result (formally defined in §6.1) a Global State Machine (GSM).

Figure 10 shows the developer’s definition of the GSM for the bank example. The State now holds all of the accounts. The transfer transition is an atomic step that reads and writes the global state. As in Figure 9b, the GSM’s state only changes—atomically—at the moment Thread 2 invokes the ghost LTS transition. Hence, regardless of low-level thread interleaving, from GSM’s global perspective, the state advances through a sequence of atomic global transitions, even as the physical values are updated asynchronously. This greatly simplifies reasoning about global correctness.

Indeed, since the GSM is a standard state machine, we can employ standard reasoning techniques honed by decades worth of research [32], including prior work automating such reasoning [19, 20]. For example, we can easily prove that account balances never go negative, or that the total amount of money across all accounts is always preserved. In both cases, the proof proceeds by showing that the property holds in an initial state, and then showing that if it holds before an atomic transition, then it also holds afterwards. For the bank example, these proofs are produced fully automatically.

In contrast, it would be impossible to talk about such global properties from within the implementation, where a given thread only ever holds at most two locks.

**Soundly Assembling the GSM.** To compose the LTS shards into the GSM’s state, the IronSync developer must declare a datatype that can hold one or more shards. §6.1 discusses the formal rules the datatype must obey, but a common pattern is to use a (partial) dictionary from an application-specific identifier (e.g., an account ID) to the corresponding shard. The developer then proves the soundness of each LTS

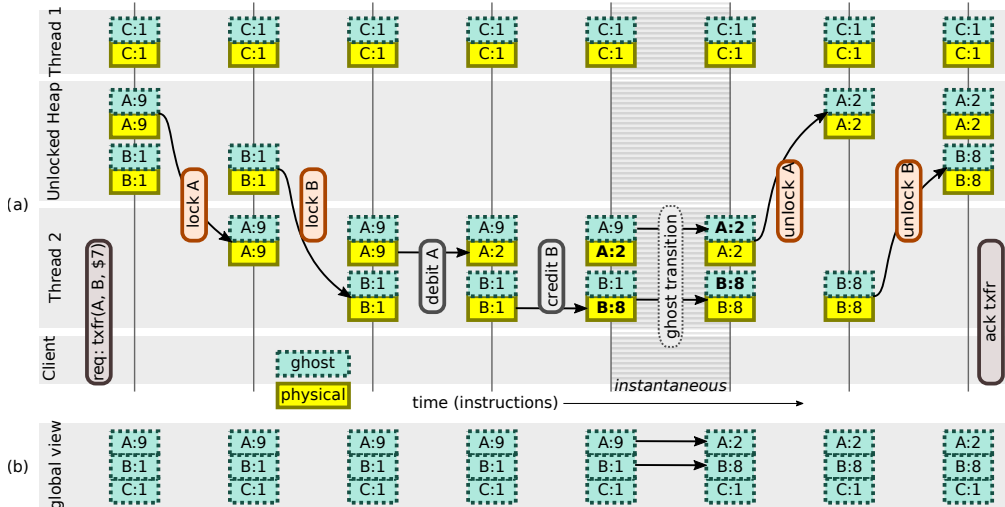


Figure 9: (a) Ghost values (dashed cyan boxes) travel alongside physical values (solid yellow). Ghost values are atomically updated according to the LTS rules (see §3.3.2 for details). (b) LTS transitions, in turn, are abstracted into the GSM (§3.3.3).

State: {accounts: map<AccountId, Balance>}

```
atomic transition transfer(A, B, x):
  if state.accounts[A] ≥ x {
    state.accounts[A] -= x;
    state.accounts[B] += x;
  }
```

Figure 10: Bank GSM

transition, with respect to this dictionary. Oversimplifying, this proof proceeds as follows. We imagine starting with a dictionary that contains at least the transition’s incoming shards (in Figure 7, the initial account information for A and B), and possibly some others as well. We remove the *incoming* shards from the dictionary and hand them to the localized transition. We then try to combine the *outgoing* shards with the dictionary. If this results in a well-formed dictionary (i.e., no keys are duplicated), the transition is valid and can be lifted to an atomic transition of the GSM (as shown in Figure 10).

## 4 Advanced IronSync Techniques

Verification of real concurrent programs often has additional challenges beyond those of our “toy” banking example. This section illustrates, via examples, IronSync’s solutions to two situations that arise in a real concurrent system:

- An abstraction of the program state (like the GSM) might still not be abstract enough to be a useful specification.
- Developers employ custom synchronization tools (beyond simple locks), plus optimizations like read-sharing.

We apply these solutions to the complex case studies in §5.

### 4.1 Specification via Refinement

For trivial programs like our bank, one might accept an invariant as the definition of correctness. For substantial programs, we prefer to express correctness via a trusted specification that precisely defines the program’s expected behavior, and then prove that the implementation refines it; i.e., every execution

Shard: {idx: nat, entry: (Key, Value)?}

```
localized transition insert(key, value):
  for some (i, j, k̄, v̄)
    where i = hash(key) and key ∉ {ki, ..., kj}
  pre-state shards:
    { for m = i .. j | {idx: m, entry: (km, vm) } }
    {idx: j+1, entry: null} // First empty slot
  post-state shards:
    { for m = i .. j | {idx: m, entry: (km, vm) } }
    {idx: j+1, entry: (key, value)} // Holds (key, value)
```

Figure 11: A Hash Table LTS Transition that inserts a new (key, value) pair at index j+1; the other shards are unmodified, but serve to justify that j+1 is the correct index.

of the implementation is an execution of the spec.

A hash table’s spec, for example, is a simple dictionary, succinctly expressible in 10-20 lines. Its implementation obeys the spec while providing good performance. For instance, a Robin Hood Hash Table (RHHT) [10] stores key-value pairs in an array and locates keys via *linear-probing* [30]: given a key, probing starts with the key’s hash index and continues sequentially until the key or an empty slot is found.

To exploit concurrency, a developer might add multiple threads and create a lock for each slot in the array. A straightforward concurrency strategy would have a thread lock the entire range of slots needed for each linear probe, complete its operation, and then release the locks. To express this strategy in IronSync, per §3, the developer defines an LTS (Figure 11) with shards at the granularity of the locking scheme: each shard of the LTS represents a single array slot.

Once the LTS is proven sound, the developer uses IronSync to reassemble the LTS into the GSM comprising the full sequence of optionally-occupied slots. They then prove that the GSM refines the spec by establishing invariants. One RHHT invariant is that each key in the table can be found in a contiguous range of non-empty slots starting from the key’s hash index. Proving refinement via such invariants is



straightforward using standard techniques [32] previously encoded in Dafny [19, 20].

## 4.2 Lower-Level Memory Primitives

The previous examples are built from locks, which help maintain data safety. In practice, many advanced concurrent systems do not use locks, but rather custom synchronization tools built from lower-level primitives. Supporting such advanced systems is a core IronSync goal, and hence IronSync makes these lower-level primitives its base and then verifiably constructs locks and other synchronization tools from them. In this section, we introduce IronSync’s primitives and, as a warm-up, see how they let us verify a basic mutual-exclusion lock.

Consider a lock implemented with two fields: a boolean *flag* indicating whether the lock is taken, and a slot for the *data* being protected by the lock. Defining operations on these fields must be done in terms of a *memory-ordering model*, which dictates when different threads may disagree on the ordering of reads and writes. Developers must take care to use special, slower instructions to synchronize threads when necessary, and such subtleties are notoriously difficult to handle correctly, especially since the details depend on the hardware platform (e.g., x86-TSO [45] or ARM [1]).

IronSync’s memory model is based on the C++11 memory model [5, 7], which abstracts over these hardware differences by providing a distinction between *non-atomic* memory (the most common, “normal” memory) and *atomic* memory. Non-atomic memory access compiles to fast instructions, while atomic memory (depending on how it is used) may compile to slower instructions, possibly involving memory fences. To make this dichotomy sound, the C++ model requires all non-atomic accesses be data-race-free (a burden placed on the programmer); however, the atomic memory allows contended access. In the lock example, multiple threads might contend to access the *flag*, but the thread that wins will have exclusive access to the *data* field, making its accesses data-race-free.

IronSync supports data-race-free non-atomic memory and sequentially-consistent atomic memory. Specifically, it takes advantage of the C++11 memory model’s *DRF-SC property*, which states that if all non-SC memory accesses are data-race-free, then the entire execution is sequentially-consistent. By allowing data-race-free memory for the common cases, IronSync takes advantage of much of the speed afforded by modern hardware, although it does not take advantage of the weaker atomic memory orderings (e.g., *release-acquire* ordering or *relaxed*).

In particular, IronSync supports these two modes of shared-memory-access through two of its trusted primitives, `Atomic` for word-sized atomic memory and `Cell<T>` for non-atomic memory storing arbitrary types *T*. To ensure that access to a shared `Cell<T>` is data-race-free, IronSync requires a thread to own a special ghost object of type `Permission<T>` for reading and writing. Meanwhile, IronSync treats the sequentially-

consistent atomics as if they were “virtual locks” that can be unlocked for a single atomic operation; they can then use lock invariants, as before, to verify code that manipulates ghost objects in the virtual lock. `Atomic` supports common atomic operations, like compare-and-swap and atomic addition.

With these tools, the developer can verify a lock as follows: they declare the *flag* field as an `Atomic` and the *data* field as a `Cell`. They store the ghost `Permission` object for the `Cell` in *flag*’s virtual lock. By reading and writing to *flag* (e.g., with an atomic compare-and-swap), threads can transfer ownership of the `Permission`, allowing them to access the *data* field in a data-race-free manner. This process is verified by IronSync, which checks that the invariant on the virtual lock is maintained.

## 4.3 Read Sharing

Crucially, data-race-freedom does not preclude all simultaneous access. While it prohibits a write from occurring simultaneously with a read or another write, it does permit multiple simultaneous readers. This read-sharing is crucial for performance in many applications; however, to make use of it, the developer must still ensure that threads obey some single-writer, multiple-reader protocol. In such a protocol, the developer ensures that there can be a single writer or multiple readers at any given time, but never both at the same time (and of course never more than one writer).

The challenge with read-sharing protocols is that there is no optimal way of accounting for the shared state. For example, a particularly common protocol uses reference-counting, e.g., in a reader-writer lock, but even here, there is no universal way to implement a reader-writer lock. Our case studies (§5), for example, employ two different custom-built reference-counting-based locks, and locks aren’t even the end of the story. Our NR case study (§5.1) uses a lock-free cyclic buffer, where multiple threads share read-access to entries, and where the safety is guaranteed by a protocol of head and tail pointers.

In IronSync, the developer can implement and verify a read-sharing protocol, including any of the above, by designing a particular kind of LTS, which we call a *guard protocol*, and proving that it enforces safe access to shared state. A guard protocol is an LTS whose state has an explicit notion of a stored (ghost) object, along with a notion of depositing and withdrawing that object. Intuitively, the program begins with a unique reference to an object (e.g., the ghost `Permission` for a `Cell` – see §4.2). To create read-shared references that it can give to other threads, the program “deposits” the object into the guard protocol, and in exchange it can obtain one or more *guards*. A guard is simply an LTS state shard that acts as a “witness” that the object has been deposited (and not yet withdrawn). Once all the guards (i.e., read-shared references) are returned, IronSync allows the program to “withdraw” the reference from the LTS and use it once again for mutation.

To demonstrate the soundness of their guard protocol, the library developer must show that it satisfies two obligations.

First, they show that guard shards only exist when an object has been deposited (and not yet withdrawn). This prevents the library from synthesizing bogus read-shared references. Second, they show that the LTS’ withdrawal transition only occurs when an object is in fact deposited and there are no outstanding guards.

Once the guard protocol is proven sound, IronSync provides the library developer with an extended version of the trusted shard API from §3.3.2. Recall that the standard API consumes and produces owned shards. The API for guard protocols, however, allows a thread holding a read guard to acquire a shared version of the protected data; e.g., a shared `Permission` for a `Cell`, which the `Cell` API requires for read access to its concrete memory, but which doesn’t suffice to use the API for writing. Hence, the developer can ergonomically manipulate shared data using Dafny’s shared variables, with the assurance that all accesses are data-race free.

Crucially, IronSync’s general approach to read sharing enables a developer to devise protocols that are drastically different from a read/write lock. For example, in the cyclic buffer (§5.1), threads read entries (via ghost guard objects) and use a head pointer to indicate when they are done; other threads look at these head pointers to determine when it is safe to garbage collect the entries and overwrite them (requiring a withdraw).

## 5 Case Studies

IronSync, we have seen, comprises a collection of tools: (ghost) ownership types, LTS abstraction, state-machine refinement, and automated verification. To test that this collection suffices to verify modern production-scale systems (i.e., systems notable for their performance, which they achieve through non-trivial concurrency patterns), we select two such systems and produce verified implementations within IronSync. These particular systems were chosen, in part, because there was independent interest in verifying their correctness from the systems’ designers. We compare our case studies to those in prior work in §9.

By producing implementations that match the originals in design and performance (§7), we show that writing a system in IronSync does not sacrifice performance-critical concurrency patterns. Of course, our implementations are not identical to the originals: ours are written in Dafny (and compiled to machine-generated C++ code), and they make a few minor deviations from the originals (§7.2). Nonetheless, the exercise does, as a bonus, yield some insight into the originals (§7.3).

**Overview.** Both case studies are complex; for each, correctness depends on myriad interlocking moving parts. Hence, we show how an IronSync developer divides the proof work into manageable subtasks, and chooses the right IronSync tool for each.

Specifically, NR (§5.1) shows how to pull together all the IronSync features discussed earlier. With SplinterCache (§5.2), we also verify the program’s use of an external disk.

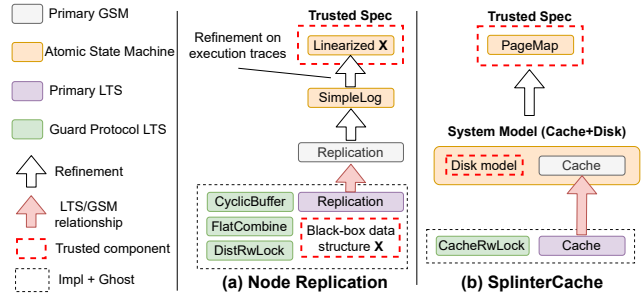


Figure 12: Proof architecture of our case studies.

**Common Architecture.** Each case study follows a similar high-level structure. Each has a *primary LTS* (and a corresponding GSM), which is used, via state-machine refinement (§4.1), to establish that the program meets its specification. In addition, each program uses complex synchronization logic that makes data safety challenging, so each case study also uses several secondary LTSes as guard protocols (§4.3). Figure 12 summarizes the architectures for the case studies in terms of these components.

### 5.1 Node Replication NR

NR [9] is a concurrency library that transforms a black-box, sequential data structure into a linearizable, NUMA-aware concurrent version. NR works by replicating the sequential data structure on each NUMA node, using an operation log to maintain consistency. Replicas benefit from read concurrency using a reader-writer lock designed to minimize reader contention [50] and from write concurrency using *flat combining* [22], which batches operations from multiple threads to be executed by a single *combiner* thread. The combiner appends the batched operations to the log; other nodes read the log and update their local replica copies. The original, unverified NR implementation is ~1000 lines of Rust. NR has recently been adopted by NrOS [6], which uses it to implement scalable versions of a wide range of OS subsystems.

**Verification Objective.** Our verified implementation also takes a user-provided black-box data structure, this time one with a functional spec. Verified NR produces the replicated data structure *and* a proof that this replicated system meets the same functional spec *linearizably*.

**Proof Overview.** As in our earlier examples, we can coarsely divide our tasks into data safety and logical correctness. In some places, the reference implementation uses Rust’s unsafe code, so these parts pose challenges for our verified implementation. In Rust, unsafe means that the code foregoes Rust’s usual safe aliasing checks and places the burden of correctness on the programmer. In IronSync, this means that we cannot (solely) rely on the ownership type system to ensure data safety. Luckily, IronSync has a verified alternative: the guard protocol.

As an example, consider the cyclic buffer at the center of NR’s coordination, used to broadcast messages from one node to all other nodes. Using a stringent protocol of head and tail

pointers, NR ensures that each node reads each message after it is written but before it is garbage collected, and further, that these reads and writes are properly synchronized. Notice how this custom protocol of head and tail pointers is used in place of a general utility for safe data access (like a mutex). Figure 13 shows pseudocode for parts of this protocol, delineating the read sections and write sections for buffer messages. Data safety, here, approximately amounts to saying that the write section never overlaps any another section. This read-sharing pattern is exactly what guard protocols are designed to support. We can construct such a protocol by identifying the instructions relevant to data safety (marked ★ in Figure 13) and abstracting them into an LTS.

NR has two more places where we need to do something similar. One is a specialized lock that protects the per-node replicas. These locks are designed with multiple reference counters to reduce thread contention; using a guard protocol, we can verify the lock and provide an API similar to the lock API discussed in §4.3. Finally, NR uses a flat-combining algorithm, and so we must reason about the synchronization of data between a client thread and a combiner thread.

With the three guard protocols, DISTRWLOCK, CYCLICBUFFER, and FLATCOMBINE to “patch up” holes in the ownership type system, the only remaining task is to prove a linearizable specification. We use an LTS, REPLICATION, to track (i) all in-progress updates and reads, (ii) the states of node-local replicas, (iii) the full history of event messages, and (iv) version numbers for the replicas. Since so much of the synchronization logic is already handled by the guard protocols, the LTS abstraction is dramatically simpler than the NR system as a whole: we have relegated entire subsystems to being “mere implementation details.”

Now, from REPLICATION, we can construct the GSM as an abstraction of the global system behavior. Next, we need to establish a state machine refinement to a linearizable specification. This is challenging because REPLICATION has *future-dependent linearization points*; hence, we cannot prove that REPLICATION refines a linearized state machine with a single-state abstraction function. Instead, we have to prove a theorem operating over arbitrary execution traces. To simplify this task, we split it into two steps: first we prove, via a single-state abstraction function, that REPLICATION’s GSM refines a simpler state machine, SIMPLELOG, which only tracks a log and the index of the latest linearized operation. Abstracting out replicas makes SIMPLELOG much easier to analyze, and the theorem over execution traces becomes tractable.

Notice that in Figure 13, the CYCLICBUFFER and REPLICATION subsystems are heavily interleaved, with some instructions even being part of both. We discuss this in §8.

## 5.2 SplinterCache

*SplinterCache* is a production-grade in-memory page cache used by the key-value store SplinterDB [14], which was created for use in VMware’s vSAN [49]. SplinterCache is built

```

fn append(ops):
  tail := globalTail;
  if tail + ops.length > globalHead + SIZE:
    wait and retry;
  compare_and_swap(globalTail, tail, tail + ops.length); ★ ■

  for i in 0 .. ops.length:
    j := (tail + i) % SIZE;
    live_bit := ((tail + i) / SIZE) % 2;
    log[j].op = ops[i]; // Non-atomic write ★
    log[j].alive = live_bit; // Synchronizing write ★

fn dispatch():
  head := nodeLocalVars.localHead; ★ ■
  tail := globalTail; ★ ■
  for i in head .. tail:
    j := i % SIZE;
    live_bit := (i / SIZE) % 2;
    wait until:
      log[j].alive == live_bit; // Synchronizing read ★
    op := log[j].op; // Non-atomic read ★
    applyUpdateToReplica(op);
    atomic_max(maxReplicaVersion, tail);
    nodeLocalVars.localHead := tail; ★ ■

```

Figure 13: Pseudocode of key NR algorithms, omitting ghost shards. Shared variables are **bold** for atomics and *italics* for non-atomics. ★ is code relevant to CYCLICBUFFER; ■ to REPLICATION. Ranges show where it is safe to read and write `log[j].op`; CYCLICBUFFER proves disjointness.

for loads where it may have 100GiB of RAM available and for use with low-latency IO devices. Clients can acquire a lock (reader or writer) on a disk page by its address, and the cache abstracts the details from the client. Internally, it loads that page into memory if necessary, and it handles writeback and eviction. Its optimizations include prefetching and batched IOs. It attempts to flush pages to disk before they need to be evicted. The reference code is ~ 2000 lines of C.

**Verification Objective.** We characterize the behavior of the cache operating together with an external disk, using a trusted model of the disk. Our high-level spec, PAGEMAP, maps each block address to two 4KiB pages, an on-disk and an in-memory version. The system may nondeterministically overwrite the on-disk version with the in-memory one.

A cache makes weaker promises than a key-value store or file system, which might offer snapshot consistency. However, this doubled-up mapping spec still constrains behavior in the event of a crash, demonstrating how IronSync programs can integrate with prior approaches to verifying crash safety [19].

**Proof Overview.** Once again, we divide our proof into data safety and logical correctness.

The SplinterCache uses a complex locking scheme to protect the in-memory cached pages. The cache implementation needs to acquire a write lock to write to a page or a read lock to read from a page, but there are some subtleties: even if the client intends to only read a page, the cache may still need to load it from disk, which means writing the contents into memory, which requires a write lock on the memory page.

For efficiency’s sake, the locking protocol has a variety of special states for handling situations like the above. More specifically, the lock has bit flags for states (not all mutually exclusive): (i) `WriteBack`: the page is being written to disk, effectively an extra read-lock; (ii) `Loading`: the page is being



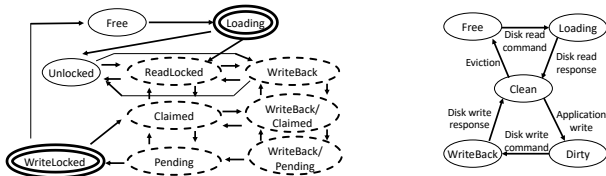


Figure 14: (a) Left, transitions for the lock status of a single cache entry in CACHERWLOCK. Dashed ovals represent read-locked states; double ovals represent write-locked states. (b) Right, transitions for the status of a single cache entry in CACHE. Both figures are simplified abstractions.

loaded from disk; (iii) Free: this entry is not assigned to a disk page; and (iv) Claimed: the claiming thread has the exclusive right to upgrade a read lock to a write lock. These states allow optimizations; e.g., an entry marked Free cannot be locked, so a thread that loads a page into a free entry can skip the usual check that there are no readers. Even beyond these states, the lock has the same multiple-reference-counters optimization as NR’s DISTRWLOCK described earlier. Figure 14a summarizes all the states.

As usual, we handle the reader-writer lock with a guard protocol. This leaves logical correctness, which we prove with the CACHE LTS. It tracks the information needed to prove consistency properties between cache and disk; e.g., it maintains a two-way mapping between cache entries (indexed by *entry numbers*) and disk pages (indexed by *page numbers*). It tracks the status of each entry, which may be either *empty* (i.e., not corresponding to any page), *loading*, *clean*, *dirty*, or *writeback-in-progress*. This is summarized in Figure 14b.

Next, we can construct the CACHE GSM as a sound abstraction of our implementation. We then apply previous techniques [19] to integrate CACHE with a (trusted) model of the asynchronous disk. This yields a state machine CACHE+DISK which abstracts the behavior of the entire system. Here, we prove relevant invariants: the bidirectional mapping is self-consistent; outstanding IO requests agree with the *loading* and *writeback* statuses; an in-memory clean page matches the on-disk page. We finally prove that CACHE+DISK refines PAGEMAP as the high-level specification.

## 6 Formalism and Implementation

IronSync’s Trusted Computing Base comprises the following:

- A trusted programming language and verifier.
- A trusted library of shared-memory primitives (§4.2).
- A trusted library of formal definitions and axioms for LTSes, guard protocols, and state machine refinement.

**The Language.** IronSync is built on Linear Dafny [35]; we added ghost ownership types to its existing (non-ghost) ownership types, and we also supply `Atomic` and `Cell` types (§4.2) for shared memory. To prevent unsoundness with concurrent threads, IronSync disallows Dafny’s traditional support for aliasable mutable objects.

IronSync code compiles via Dafny’s C++ backend, and uses `std::atomic` to implement IronSync’s `Atomic`. Therefore, the C++ compiler is also part of the trusted toolchain, notably including its mapping from C++’s memory ordering model to the hardware’s. Specifically, we rely on the compiler to insert memory fences appropriately for any `Atomic` memory locations, thereby providing the sequential consistency guarantees for the language runtime.

### 6.1 Formal Definitions

We briefly summarize the mathematical formalism underpinning IronSync’s LTS (§3.3.1) and the guard protocol (§4.3). These definitions are exposed to the IronSync developer via an axiomatically trusted library in Dafny.

IronSync introduces LTSes to formalize the idea that a transition updates and depends on only a portion of the state, while the rest of the state is *irrelevant*. This formalization encodes shards as elements of a *monoid*, an established tool from concurrency reasoning from separation logic [26].

**Definition 1 (LTS: Localized Transition System)** *A Localized Transition System is a triple  $(M, \text{Init}, \tau_{\text{local}})$ . Here,  $M$  is a commutative monoid, that is, a set with a composition operator  $(\cdot) : M \times M \rightarrow M$  which is associative and commutative, and with a unit element  $\epsilon \in M$  (i.e.,  $\forall m \in M. m \cdot \epsilon = m$ ). Meanwhile,  $\text{Init} : M \rightarrow \text{bool}$  represents valid initial states, and  $\tau_{\text{local}} : M \times M \rightarrow \text{bool}$  is a “local transition function.”*

This essentially says that an element  $m \in M$  represents partial information about a state of the system, while the composition of two elements gives us a way to combine the partial information about different components. Thus a transition  $\tau$  makes sense even on pieces of partial information. In the bank example discussed at the end of §3.3.3, each  $m$  is a (partial) dictionary from account IDs to account info.

The IronSync framework then defines the global state machine (GSM) in terms of the LTS by taking elements of  $m$  that represent a complete view of the system (e.g., a dictionary containing all of the bank’s accounts). Specifically, we define a transition on a complete state by splitting it in two: one part to be operated on, and one part that is irrelevant to the transition, and then performing the transition on the first part:

**Definition 2 (GSM: Global State Machine)** *Given an LTS  $(M, \text{Init}, \tau_{\text{local}})$ , we define a global transition function,*

$$\tau_{\text{global}}(s, s') \triangleq \exists d, d', e. \tau_{\text{local}}(d, d') \wedge (s = d \cdot e) \wedge (s' = d' \cdot e).$$

*We call  $(M, \text{Init}, \tau_{\text{global}})$  the Global State Machine.*

In the bank example,  $d$  would be a dictionary holding keys for the two accounts involved in a transfer, and  $e$  would be a dictionary holding all of the other accounts.

### 6.2 Guard Protocols

A Guard Protocol consists of (i) an LTS with a notion of ghost objects that may be *deposited* or *withdrawn*, and (ii) a notion of a *guard*, a state shard that locally guarantees a particular



ghost object is deposited. Concretely, a Guard Protocol is defined by its relation to the following state machine that formalizes “deposited state.”

**Definition 3 (Safety-Deposit State Machine)** *Given a set  $T$ , a Safety-Deposit State Machine is a state machine defined over the state  $T \cup \{\text{empty}\}$ , with transitions, for all  $t \in T$ :*

$$\begin{array}{lcl} \text{empty} & \xrightarrow{\text{deposit}(t)} & t \\ & & t \xrightarrow{\text{withdraw}(t)} \text{empty} \\ \text{empty} & \xrightarrow{\text{internal}} & \text{empty} \\ & & t \xrightarrow{\text{internal}} t \end{array}$$

Here,  $T$  is the set of ghost objects that can be deposited.

A developer defines an LTS for their Guard Protocol and proves it sound based on the definition below, and in exchange, IronSync gives them access to a set of ghost shards representing their protocol: a thread can deposit ghost objects from the set  $T$  into the protocol and withdraw them later. Crucially, the trusted IronSync API from §4.3 allows code that holds a guard shard to obtain a **shared** copy of the deposited value  $t$ , which allows the code to ergonomically and soundly manipulate read-shared data.

Formally, we define a Guard Protocol as follows.

**Definition 4 (Guard Protocol)** *Given a set  $T$ , a Guard Protocol is an LTS that has three transition types,  $(M, \text{Init}, \tau_{\text{local}}^{\text{internal}}, \tau_{\text{local}}^{\text{deposit}(t)}, \tau_{\text{local}}^{\text{withdraw}(t)})$ ; an invariant  $\text{Inv} : M \rightarrow \text{bool}$ ; and an abstraction function  $\text{Abs} : M \rightarrow T \cup \{\text{empty}\}$ . We define  $\tau_{\text{global}}^{\text{internal}}, \tau_{\text{global}}^{\text{deposit}(t)}, \tau_{\text{global}}^{\text{withdraw}(t)}$  of the GSM as in Def. 2.*

We say the Guard Protocol is sound if  $\text{Inv}$  is an inductive invariant on the GSM, i.e.,  $\forall$  transition labels  $\ell$ :

$$\begin{array}{l} \forall m \in M. \text{Init}(m) \implies \text{Inv}(m) \\ \forall m, m' \in M. \text{Inv}(m) \wedge \tau_{\text{global}}^{\ell}(m, m') \implies \text{Inv}(m') \end{array}$$

and if the GSM, as interpreted by  $\text{Abs}$ , refines the Safety-Deposit State Machine. Given a sound Guard Protocol, we say that  $g \in M$  is a read-guard of  $t \in T$  if,

$$\forall b \in M. \text{Inv}(g \cdot b) \implies \text{Abs}(g \cdot b) = t.$$

Here,  $\text{Abs}$  gives the GSM a notion of a “deposited object.” The read-guard condition says that in any valid global state (as given by  $\text{Inv}$ ) with  $g$  as a sub-shard,  $t$  is guaranteed to be the deposited object (as given by  $\text{Abs}$ ). This means that a thread holding the guard shard  $g$  can soundly read the shared value  $t$ , and that all such readers will read the same value.

When the IronSync user defines a new guard protocol and proves it sound, IronSync gives them access to an API to manipulate ghost shards according to the transitions, as with any other LTS. In this case, the functions that perform exchanges can also perform deposits and withdraws; furthermore, there are new functions for the read-guard objects: if  $g$  is a read-guard of  $t$ , then the user can use a **shared** ghost shard  $g$  to obtain a **shared** ghost shard  $t$ . Linear Dafny’s type system ensures that the guard reference outlives  $t$ .

Major component	trusted LOC	impl LOC	proof LOC	verif time
<b>Common Framework</b>				
LTS def. & ghost axioms	487			15 s
Memory Primitives	310			6 s
Libraries		316	3825	75 s
<b>Bank §3</b>				
Spec	17			0.7 s
LTS			262	9 s
Impl		21	16	2 s
<b>RHHT Hash Table §4.1</b>				
Spec	57			2 s
LTS			687	54 s
Refinement Proofs			168	8 s
Impl		417	1390	68 s
<b>Node Replication §5.1</b>				
Spec	104			4 s
REPLICATION LTS			2329	384 s
FLATCOMBINE LTS			649	97 s
CYCLICBUFFER LTS			1756	182 s
DISTRWLOCK LTS			633	17 s
Refinement Proofs			1291	132 s
Impl		730	1170	80 s
<b>SplinterCache §5.2</b>				
Spec	185			4 s
Disk Model and API	586			14 s
CACHE LTS			1036	159 s
CACHERWLOCK LTS			2015	86 s
Refinement Proofs			2456	372 s
Impl		1579	3163	297 s
Total	1746	3063	22846	34.7 min

Figure 15: Across all case studies the proof:code ratio is 7.5.

In addition to the formulation above, IronSync provides a more advanced version that allows multiple objects to be stored at once. This is useful for NR’s cyclic buffer §5.1, for example. Both of these formulations are proved correct, based on a set of low-level axioms for manipulating monoid-based ghost state with **shared** variables in Linear Dafny; those axioms in turn are based on a concurrent separation logic for temporary read-sharing called Burrow [18].

## 7 Evaluation

In our evaluation, we aim to answer the following questions:

- What is the verification effort for IronSync development, both by the developer and the computer verifier (§7.1)?
- Is IronSync suitable for verifying state-of-the-art systems without compromises (§7.2)?
- What does verification tell us about the original reference implementations (§7.3)?

### 7.1 Verification Effort

Verifying all four concurrent examples consumes under an hour of CPU (5 minutes real time) on an 8-core 64 GiB cloud machine. 88% of files verify in under a minute; the slowest takes less than five. The four examples comprise 2747 lines of non-ghost implementation, plus 316 of shared library.

Figure 15 shows detailed information for each case study. Within implementation files, the proof-to-code ratio is about 4:1, where the proof code includes both the manipulation of ghost shards and standard Dafny proof annotations, like preconditions and postconditions. The full system proofs augment the implementation files with LTS code and refinement proofs, raising the overall proof:code ratio to 7.5.

As two points of comparison, we consider GoJournal [12] and Armada [38] (see §9 for details). GoJournal [12] reports a 19:1 proof-to-code ratio for its shared-memory code, while Armada’s largest example takes 4.9 hours of CPU time (about 40 min. of real time) to verify 70 lines of code. These are not direct apples-to-apples comparisons: Armada and GoJournal arguably prove more substantial theorems about machine semantics. Still, verification time and developer effort have historically limited the use of verification tools, and thus IronSync constitutes a major practical improvement.

## 7.2 Case Study Fidelity

We evaluate IronSync’s expressiveness by porting our two production-level case studies, NR (§5.1) and SplinterCache (§5.2), to IronSync to confirm that IronSync does not require sacrificing performance-critical concurrency patterns. We refer to the case studies’ existing publications [9, 14] (both within the last 5 years) to justify that they can reasonably be called “state-of-the-art.” We evaluate how faithful our IronSync implementations are to the reference implementations both qualitatively and by comparing performance.

First, we report on intentional compromises we made while mimicking the reference code of NR and SplinterCache, from most significant to least. First, in some cases, Reference NR uses release-acquire atomics. IronSync does not support these, so we use sequentially-consistent atomics instead. Second, IronSync does not support callbacks, so we refactored code to avoid them, and we could not implement the secondary, callback-based APIs in SplinterCache or NR. Third, IronSync’s SplinterCache adds a runtime check in one method whose correctness was otherwise dependent on properties of SplinterDB’s allocator, which was out-of-scope.

As evidence that these artifacts otherwise meet a high degree of fidelity, we benchmark against their references, using methodology similar to the reference publications [9, 14]. Each case study has different hardware requirements.

**NR.** We evaluate NR’s performance against other locks and to its reference Rust implementation from NrOS [6]. We wrap a single-threaded radix tree with IronSync-NR, Reference-NR, or a lock, including a verified DISTRWLOCK (§5.1), an MCS lock [40], a shuffle lock [27], and the standard C++ shared mutex. The benchmark pre-populates the tree with 128M entries (using 8B keys and values) and executes *get* and *update* requests with a uniform key distribution while varying the update ratio and the number of threads.

Figure 16 shows the performance measured on a machine with 4 Xeon Gold 6252 CPUs with 24 cores per NUMA node,

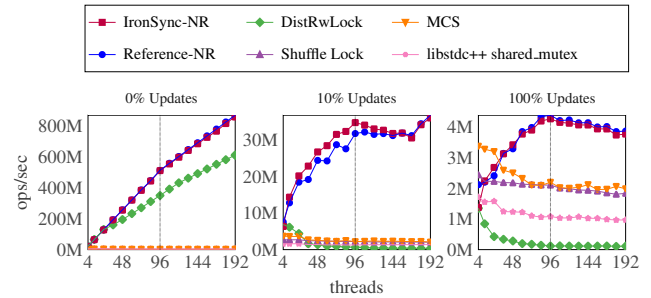


Figure 16: Comparing throughput scalability of IronSync-NR, Reference-NR, and locks. Higher is better.

totaling 96 cores and 192 hardware threads. The threads are pinned to fill up cores on a NUMA node first before moving to the next. NR adds one replica for each NUMA node, so at  $x=96$  threads, NR uses 4 replicas. Beyond 96 threads, no more replicas are added, and we begin hyperthread-sharing. IronSync-NR and Reference-NR perform similarly and generally outperform the rest, especially for read-heavy workloads.

For 0% updates, IronSync-NR, Reference-NR and the DISTRWLOCK scale linearly, but the other mechanisms suffer under lock contention. NR performs better than DISTRWLOCK due to perfect NUMA locality. With 10% updates, DISTRWLOCK’s performance drops to match the other locks, while IronSync-NR and Reference-NR benefit from flat combining. IronSync-NR outperforms Reference-NR slightly, though we do not yet know the cause.

Only at very high update rates (e.g., 100%) do MCS and shuffle locks outperform NR at low scale on one NUMA node; otherwise both NR implementations dominate. Hence, we conclude that IronSync-NR provides performance parity with Reference-NR and that it preserves NR’s replication and flat combining benefits at all scales.

**SplinterCache.** We evaluate the performance of IronSync-SplinterCache against the reference implementation both with macrobenchmarks as part of SplinterDB using the YCSB benchmark [15], and with cache-specific microbenchmarks.

Results are from a Dell PowerEdge R630 with a 28-core 2.00 GHz Intel Xeon E5-2660 CPU, 192 GiB RAM and a 960GiB Intel Optane 905p PCI Express 3.0 NVMe device.

**Macrobenchmarks.** Our YCSB configuration largely follows prior work [14]. We perform the Load, and A-F standard workloads on SplinterDB using either IronSync- or Reference-SplinterCache. Each workload uses 24B keys, 100B values and 14 threads. Run E performs 14M operations and the others each perform 69M operations, so that each workload logically reads/writes roughly 80GiB of data.

We use three target memory sizes: 4 GiB to stress eviction and IO; 20 GiB to reflect a common system configuration; and 100 GiB to stress in-memory and concurrency. Figure 17 shows that SplinterDB with IronSync-SplinterCache is always within 9% of the reference performance.

**Microbenchmarks.** We first allocate pages and flush them

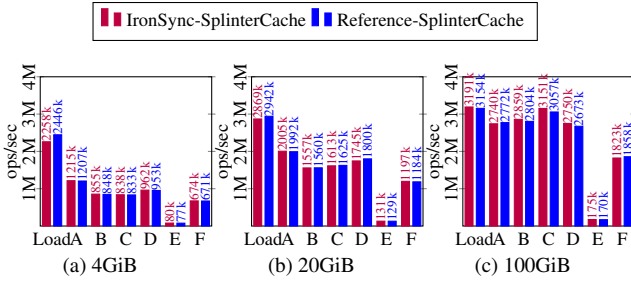


Figure 17: YCSB Benchmark. 69M ops/workload (E is 14M) with 14 threads. Y-axis is mean of 3 runs. Higher is better.

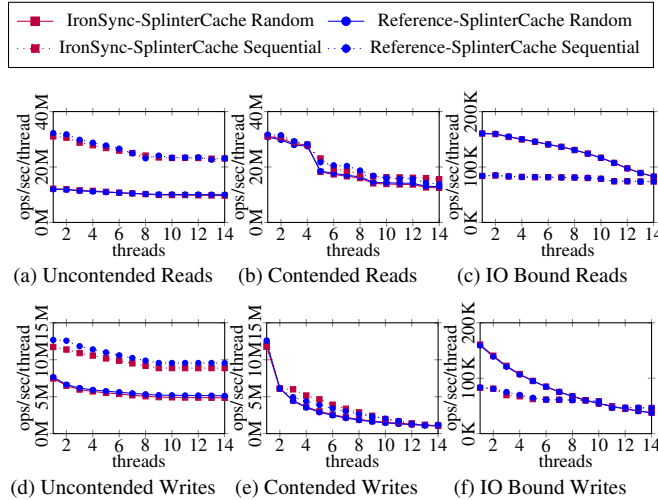


Figure 18: SplinterCache microbenchmark with a 4 GiB cache. Y-axis is mean throughput of 5 runs. Higher is better.

without evicting them. Then each thread performs a fixed number of operations, choosing pages either randomly or sequentially, then either acquiring a read lock or a write lock. We use three configurations in a 4 GiB cache: general “uncontended” in-memory, with 2 GiB of data, (Figures 18a and 18d), “contended” in-memory, with 128 KiB (32 pages) of data (Figures 18b and 18e), and “IO bound”, with 8 GiB of data (Figures 18c and 18f). IronSync-SplinterCache is within 11% of the performance of reference on all microbenchmarks.

### 7.3 Bugs and Insights

We confirmed the 3 bugs below with the original developers.

**NR.** In the reference code, we identified a bug which could cause a read-read linearizability violation between two different nodes if they took place concurrently with an update. This bug could only occur if a thread dispatched log entries during garbage collection.

This bug surfaced when we realized our first attempt at defining REPLICATION would not be linearizable. We fixed the bug by always holding the lock appropriately, and the verified implementation now puts an extra ghost shard behind the lock to represent the lock’s role in REPLICATION.

**SplinterCache.** We identified two bugs in the reference code. First was a data race on `disk_addr`, which maps cache entries to disk addresses. This race could occur when a read

lock races with both eviction and a subsequent load.

Second, the code for batching write IO did not check that `disk_addr` was the expected value after locking a page for writeback. This could result in data written to the wrong location, among other corruptions. We identified these while porting the implementation, as we realized certain ghost shards would not be available following the reference logic.

## 8 Discussion on Modularity

Concurrent systems can seem dauntingly anti-modular when they entangle low-level synchronization with high-level application logic, making the tasks of ensuring data safety and logical correctness seem inseparably intertwined. By verifying two such case studies, IronSync shows how these two levels of concern can be disentangled *within the proof*.

In NR (§5.1), for example, the `localHead` variables play two distinct roles: (i) buffer entries cannot be garbage-collected past any node’s `localHead` (relevant to CYCLICBUFFER), ensuring data safety, and (ii) `localHead` matches the version of the local replica state (relevant to REPLICATION), ensuring logical correctness. Figure 13 illustrates the overlap of these roles in two methods where the overlap is notably dense. Note that some operations might advance both state machines at the same time; however, this fact is not relevant to proofs associated with either either half.

Likewise in SplinterCache (§5.2), the `WriteBack` flag plays a role in both logical correctness (CACHE) and low-level data safety (CACHERWLOCK). The code ties these two distinct roles together by using the same flag bit: when a thread modifies the physical `WriteBack` flag, it advances *both* state machines (Figure 14), but again, this is an implementation detail to which both abstractions are agnostic.

In short, we modularize proofs of a sophisticated system by abstracting it in *multiple* ways. Difficult concurrent reasoning takes place on simplified abstractions, but IronSync ensures the abstractions compose soundly; thus proofs about the individual components say something meaningful about the whole. The implementation still ties the abstractions together with physical state, but this step is straightforward from a verification standpoint, thanks to the ownership type system and Dafny’s automation. Ultimately, this method decouples the modularity structure of the *proof* from that of the *code*.

## 9 Related Work

Logics for concurrent programs reflect different trade-offs between generality, expressiveness, modularity, and usability. IronSync strikes a balance between very general state machines at high levels of abstraction, while at lower levels leaning on language features like Hoare logic and ownership types for usability. IronSync trusts these language features instead of proving theorems directly against operational semantics, unlike work like Armada [38] or Iris [24].

Concurrent separation logic (CSL) [44] lets threads take temporary ownership of state to perform isolated reasoning.



Propositional CSL is in general undecidable [8], so tools either require manual assistance from the user, as in Iris Proof Mode [31] where the user can manually match hypotheses to goals, use incomplete heuristics and user hints, as in Diaframe [42], or solve restricted fragments, as in Viper [43] and Steel [16]. Meanwhile, IronSync encodes CSL propositions using explicit ownership in the type system. Thus, ownership is directed manually by the user, but this method lets us additionally take direct advantage of automation from standard sequential reasoning tools such as SMT solvers and the weakest-precondition-style encoding used by Dafny.

Recent CSLs are extremely sophisticated. Iris [24] and Steel [16] employ monoids to extend CSL with flexible ownership protocols, used in recent systems like Perennial [11] and GoJournal [12], and Iris can handle future-dependent linearization points with prophecy variables [25]. However, the proof rules in these systems are intricate and may be intimidating to non-experts (e.g. Iris needs the “later modality” to allow impredicative invariants, which allow Iris to express some invariants beyond what IronSync can handle directly). In contrast, IronSync aims to make these concepts approachable by integrating invariants and monoids into its ownership type system, and connecting them with state-machine refinement. As a rough comparison, IronSync’s case studies achieve a 7.7:1 proof-to-code ratio, while GoJournal [12] reports a 19:1 ratio for a comparably sized case study.

Like IronSync, Armada [38], IronFleet [20], and VeriBetrKV [19] all employ state-machine refinement. The latter two use Dafny’s Hoare reasoning for the implementation of sequential code, whereas IronSync uses it for concurrent shared-memory code. In contrast, Armada verifies concurrent code using state machine refinement throughout the entire proof stack, foregoing Hoare-style reasoning in favor of detailed, low-level state machines. This provides more expressiveness; for example, two of Armada’s case studies rely on racy memory accesses using memory ordering weaker than SC, which IronSync does not currently support. However, Armada’s expressiveness also imposes costs; e.g., Armada’s Pointers case study is 13 LoC and generates 6,997 lines of proof, while in IronSync the proof is trivial, since the correct usage of the owned pointers is automatically determined by type checking. Similarly, Armada’s Owicki-Gries counter requires 130 lines of manual proof and generates 169,270 lines of Dafny proof to verify, while in IronSync it requires 230 lines of manual proof that verify directly in 8 seconds. We studied Armada’s largest case study, a lock-free queue with 70 lines of code, and implemented an analogous queue in IronSync. Theirs requires 601 lines of proof (compared to 580 for IronSync) and 8 proof layers ( $\sim 700$  LoC), and takes 4.9 hours of CPU time to verify almost 200K lines of generated proof, versus 100 seconds of CPU time for IronSync.

Many other systems utilize ownership types. Cogent [3] and VeriBetrKV [19] use ownership types for systems verification, albeit with no shared-memory concurrency, and

with the latter introducing Linear Dafny. CIVL [21] (based mainly on reduction [37]) uses ownership types, but primarily to handle thread identifiers, not general ghost state. Rust [28, 39] uses ownership types to enforce memory safety between threads [23] but lacks verification of deeper correctness properties. GhostCell [53] (an inspiration for our `Cell`) proposes owned “ghost tokens” in Rust to express ownership of groups of objects, though only for memory safety. Tools like Prusti [4] verify single-threaded Rust programs; IronSync can help extend them to multi-threaded Rust code.

Several approaches use Dafny-style automation for concurrent reasoning. Chalice [34] is a Dafny-like language with lock invariants but no tools for global reasoning. GoJournal [12] does integrate Dafny’s sequential reasoning into a verified concurrent system, but it performs its shared-memory concurrency reasoning in Iris, so it does not leverage Dafny’s automation for *concurrency* reasoning the way IronSync does.

CertiKOS [17] and SeKVM [36, 46] encapsulate concurrent operations inside modular interfaces, where programmers write proofs about the operations directly in Coq. We expect that IronSync-style ownership could simplify these proofs.

Prior work has verified concurrent hash tables, both bucketing [13] and linear-probing [18]. Prior work has also verified flat-combining [47] and producer-consumer queues [41, 48], but we are not aware of a verified cyclic buffer like NR’s, which requires multiple consumers to read each entry.

## 10 Conclusion

IronSync offers scalable verification of concurrent shared-memory systems by factoring their complex proofs into separate concerns. It automates proofs of data safety and local logical correctness via a fast, deterministic ownership type system combined with powerful tools for *sequential* correctness. IronSync’s LTS connects these local techniques to a simplified view of the entire system, where a developer can more easily reason about global properties. Our case studies demonstrate the success of this approach and show that we can tease apart application and synchronization logic for proof purposes, even when the implementation entangles them.

## 11 Acknowledgments

Work at CMU was supported, in part, by the Alfred P. Sloan Foundation, a Google Faculty Fellowship, a gift from VMware, a grant from the Intel Corporation, and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Andrea Lattuada was supported by a Google PhD Fellowship.

We thank Mihai Budiu, Manos Kapritsos, Jay Lorch, and Oded Padon, along with the anonymous reviewers and our shepherd, Eddie Kohler, for helpful feedback. We also thank Rob Johnson for discussions on SplinterCache.



## A Artifact Appendix

### Abstract

Our artifact contains everything needed to verify the IronSync projects with Linear Dafny and run the benchmark experiments.

### Scope

Our artifact can be used to reproduce the results in Section 7, specifically:

- The table in Figure 15, with line count information and verification times.
- Other figures, such as proof-to-code ratio, mentioned in Section 7.1.
- Benchmark results in Section 7.2, specifically, the claim that for both NR and SplinterCache, the performance of the IronSync applications are comparable to their corresponding reference implementations.

### Contents

The artifact contains:

- Linear Dafny source for the IronSync framework.
- Linear Dafny source for the case studies mentioned in the paper (the bank, the hash table, SplinterCache, and NR)
- Our modified version of Linear Dafny needed to run IronSync.
- The open-source reference implementation of NR.
- The open-source reference implementation of SplinterDB (which includes SplinterCache).
- A Docker container with all Dafny dependencies.
- A benchmark harness for each.

### Hosting

The artifact is hosted at <https://github.com/secure-foundations/ironsync-osdi2023>, commit d361111cbc87b5573d14975227de845e8a717ca5. See the README.md file for instructions.

### Requirements

The artifact requires x86 Ubuntu. (Note that, although our artifact includes a Docker container, which may be used on other platforms, the Docker container is only used for running Linear Dafny; it cannot be used for running the benchmarks.)

The ideal hardware for the NR benchmarks is a NUMA machine with 96 cores. The benchmarks in our paper were run on a machine with 4 Xeon Gold 6252 CPUs with 24 cores per NUMA node. However, a machine with fewer cores should still be able to reproduce our graphs up to a certain number of threads.

The ideal hardware for the cache benchmarks is a machine with a low-latency storage device, such as an Intel 905P Optane SSD. The machine also needs at least 100GiB to run the largest benchmark. The benchmarks in our paper were

run on a Dell PowerEdge R630 with a 28-core 2.00 GHz Intel Xeon E5-2660 CPU, with 192 GiB RAM.

If the ideal hardware is not used, you will not be able to reproduce the exact performance characteristics of our paper, though we still expect to see that the IronSync implementations perform comparably to the reference implementations. Our artifact contains additional recommendations for selecting hardware.

### References

- [1] ALGLAVE, J., FOX, A., ISHTIAQ, S., MYREEN, M. O., SARKAR, S., SEWELL, P., AND NARDELLI, F. Z. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP)* (2009).
- [2] ALGLAVE, J., MARANGET, L., SARKAR, S., AND SEWELL, P. Fences in weak memory models. In *Proceedings of Computer Aided Verification (CAV)* (2010).
- [3] AMANI, S., HIXON, A., CHEN, Z., RIZKALLAH, C., CHUBB, P., O'CONNOR, L., BEEREN, J., NAGASHIMA, Y., LIM, J., SEWELL, T., TUONG, J., KELLER, G., MURRAY, T., KLEIN, G., AND HEISER, G. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).
- [4] ASTRAUSKAS, V., MÜLLER, P., POLI, F., AND SUMMERS, A. J. Leveraging Rust types for modular specification and verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (October 2019).
- [5] BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. Mathematizing C++ concurrency. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2011).
- [6] BHARDWAJ, A., KULKARNI, C., ACHERMANN, R., CALCIU, I., KASHYAP, S., STUTSMAN, R., TAI, A., AND ZELLWEGER, G. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021).
- [7] BOEHM, H.-J., AND ADVE, S. V. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2008).
- [8] BROTHERSTON, J., AND KANOVICH, M. Undecidability of propositional separation logic and its neighbours. *J. ACM* 61, 2 (Apr. 2014).

- [9] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).
- [10] CELIS, P. *Robin Hood Hashing*. PhD thesis, University of Waterloo, CAN, 1986.
- [11] CHAJED, T., TASSAROTTI, J., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2019).
- [12] CHAJED, T., TASSAROTTI, J., THENG, M., JUNG, R., KAASHOEK, M. F., AND ZELDOVICH, N. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2021).
- [13] CLAUSEN, E. Verifying hash tables in Iris. Master’s thesis, Aarhus University, 2017.
- [14] CONWAY, A., GUPTA, A. K., CHIDAMBARAM, V., FARACH-COLTON, M., SPILLANE, R. P., TAI, A., AND JOHNSON, R. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2020).
- [15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing* (2010).
- [16] FROMHERZ, A., RASTOGI, A., SWAMY, N., GIBSON, S., MARTÍNEZ, G., MERIGOUX, D., AND RAMANANANDRO, T. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages* 5, ICFP (August 2021).
- [17] GU, R., SHAO, Z., CHEN, H., WU, X., KIM, J., SJÖBERG, V., AND COSTANZO, D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (2016).
- [18] HANCE, T., HOWELL, J., PADON, O., AND PARNO, B. Burrow: Custom read/write permissions for custom ghost state in concurrent separation logic. Tech. Rep. CMU-CyLab-21-002, Carnegie Mellon University, Cylab, Nov. 2021.
- [19] HANCE, T., LATTUADA, A., HAWBLITZEL, C., HOWELL, J., JOHNSON, R., AND PARNO, B. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020).
- [20] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2015).
- [21] HAWBLITZEL, C., PETRANK, E., QADEER, S., AND TASIRAN, S. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of Computer Aided Verification (CAV)* (2015).
- [22] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the synchronization-parallelism tradeoff. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2010).
- [23] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018).
- [24] JUNG, R., KREBBERS, R., JOURDAN, J.-H., BIZJAK, A., BIRKEDAL, L., AND DREYER, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).
- [25] JUNG, R., LEPIGRE, R., PARTHASARATHY, G., RAPOPORT, M., TIMANY, A., DREYER, D., AND JACOBS, B. The future is ours: Prophecy variables in separation logic. *Proceedings of the ACM Programming Languages* 4, POPL (Jan. 2020).
- [26] JUNG, R., SWASEY, D., SIECZKOWSKI, F., SVENDSEN, K., TURON, A., BIRKEDAL, L., AND DREYER, D. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).
- [27] KASHYAP, S., CALCIU, I., CHENG, X., MIN, C., AND KIM, T. Scalable and practical locking with shuffling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2019).
- [28] KLABNIK, S., AND NICHOLS, C. *The Rust Programming Language*. No Starch Press, USA, 2018.
- [29] KLABNIK, S., NICHOLS, C., AND RUST COMMUNITY. The Rust Programming Language. <https://doc.rust-lang.org/book/>.

- [30] KNUTH, D. E. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.
- [31] KREBBERS, R., TIMANY, A., AND BIRKEDAL, L. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.* 52, 1 (Jan. 2017), 205–217.
- [32] LAMPORT, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [33] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).
- [34] LEINO, K. R. M., MÜLLER, P., AND SMANS, J. Verification of concurrent programs with Chalice. In *Proceedings of Foundations of Security Analysis and Design (FOSAD)* (2009).
- [35] LI, J., LATTUADA, A., ZHOU, Y., CAMERON, J., HOWELL, J., PARNO, B., AND HAWBLITZEL, C. Linear types for large-scale systems verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (November 2022).
- [36] LI, S.-W., LI, X., GU, R., NIEH, J., AND HUI, J. Z. A secure and formally verified Linux KVM hypervisor. In *Proceedings of the IEEE Symposium on Security and Privacy* (2021).
- [37] LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18, 12 (1975).
- [38] LORCH, J. R., CHEN, Y., KAPRITSOS, M., MA, H., PARNO, B., QADEER, S., SHARMA, U., WILCOX, J. R., AND ZHAO, X. Armada: Automated verification of concurrent code with sound semantic extensibility. *ACM Transactions on Programming Languages and Systems* 44, 2 (June 2022).
- [39] MATSAKIS, N. D., AND KLOCK, F. S. The Rust language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104.
- [40] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (Feb. 1991).
- [41] MÉVEL, G., AND JOURDAN, J.-H. Formal verification of a concurrent bounded queue in a weak memory model. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021).
- [42] MULDER, I., KREBBERS, R., AND GEUVERS, H. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2022).
- [43] MÜLLER, P., SCHWERHOFF, M., AND SUMMERS, A. J. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)* (Berlin, Heidelberg, 2016).
- [44] O’HEARN, P. W. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1–3 (Apr. 2007).
- [45] OWENS, S., SARKAR, S., AND SEWELL, P. A better x86 memory model: x86-TSO. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (Aug. 2009).
- [46] TAO, R., YAO, J., LI, X., LI, S.-W., NIEH, J., AND GU, R. Formal verification of a multiprocessor hypervisor on Arm relaxed memory hardware. In *Symposium on Operating Systems Principles (SOSP)* (2021).
- [47] TURON, A., DREYER, D., AND BIRKEDAL, L. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)* (2013).
- [48] TURON, A., VAFEIADIS, V., AND DREYER, D. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)* (2014).
- [49] VMWARE. What is VMware vSAN? <https://www.vmware.com/products/vsan.html>, 2021.
- [50] VYUKOV, D. Distributed reader-writer mutex. <http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex>, 2011.
- [51] WADLER, P. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods* (1990).
- [52] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Nov. 2013).
- [53] YANOVSKI, J., DANG, H.-H., JUNG, R., AND DREYER, D. GhostCell: Separating permissions from

data in Rust. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021).





# Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs

Qiang Li<sup>◇</sup>, Yixiao Gao<sup>‡</sup>, Xiaoliang Wang<sup>‡</sup>, Haonan Qiu<sup>◇</sup>, Yanfang Le<sup>‡</sup>, Derui Liu<sup>◇</sup>, Qiao Xiang<sup>\*</sup>,  
Fei Feng<sup>◇</sup>, Peng Zhang<sup>◇</sup>, Bo Li<sup>◇</sup>, Jianbo Dong<sup>◇</sup>, Lingbo Tang<sup>◇</sup>, Hongqiang Harry Liu<sup>◇</sup>, Shaozong Liu<sup>◇</sup>,  
Weijie Li<sup>◇</sup>, Rui Miao<sup>◇</sup>, Yaohui Wu<sup>◇</sup>, Zhiwu Wu<sup>◇</sup>, Chao Han<sup>◇</sup>, Lei Yan<sup>◇</sup>, Zheng Cao<sup>◇</sup>, Zhongjie Wu<sup>◇</sup>,  
Chen Tian<sup>‡</sup>, Guihai Chen<sup>‡</sup>, Dennis Cai<sup>◇</sup>, Jinbo Wu<sup>◇</sup>, Jiaji Zhu<sup>◇</sup>, Jiasheng Wu<sup>◇</sup>, Jiwu Shu<sup>\*</sup>  
<sup>◇</sup>Alibaba Group, <sup>‡</sup>Nanjing University, <sup>‡</sup>AMD, <sup>\*</sup>Xiamen University

## Abstract

Datacenter applications have been increasingly applying RDMA for ultra-low latency and low CPU overhead. However, RDMA-capable NICs (RNICs) of different vendors or different generations of the same vendor do not cooperate well, which could cause bandwidth imbalance in the production network and introduce new root causes of the PFC storms. Our key observation is that although the data path functions of heterogeneous RNICs follow the same RoCEv2 specifications, their control path functions are vendor and version specific. To this end, we propose Flor, an open framework that provides a unified hardware data plane atop heterogeneous RNICs and a flexible software control plane running over host CPUs or NPUs of RNICs and DPUs. The hardware plane requires no changes to current specifications. The software plane on-loads congestion control and reliability management in the large-scale lossy Ethernet with no PFC dependency. We implemented and evaluated Flor in both testbed and production clusters over Intel E180, Mellanox CX-4 and CX-5 and Broadcom RNICs. Experiments show that Flor achieves comparable performance to vanilla RDMA in many scenarios, including 1/4096 packet loss, 6000:1 incast, and large-scale cross-pod communication. Flor mitigates the performance gap of CX-4 and CX-5 RNICs from 24.3% to 1.3% when they are deployed together.

## 1 Introduction

Remote Direct Memory Access (RDMA) over Converged Ethernet has been widely deployed in datacenters [3, 5, 11, 14, 30]. It provides low latency and high throughput for many applications, *e.g.*, key-value store [21, 35], distributed transactions [8, 55], distributed memory [9, 56], remote procedure call (RPC) [20, 22, 47], storage systems [11], graph computing [43] and machine-learning systems [29].

With the increasing deployment of RDMA, modern datacenters adopted RDMA-capable NICs (RNICs) of different generations and vendors, *e.g.*, Mellanox ConnectX-(CX-4)/5/6 [49, 50, 52], BlueField [51], Intel E810 [17], and cloud-provider customized RNICs [10, 12, 42]. On the one hand,

adopting more than one vendor avoids vendor lock-in, *i.e.*, relying on devices of a particular vendor, which is a serious risk during global supply chain crises such as the COVID-19 pandemic [18, 45]. On the other hand, the disaggregated deployment of storage and computation systems separates the back-end services from the front-end services into different clusters, where each cluster can host different types of RNICs.

The coexistence of heterogeneous network devices in datacenters introduces new challenges [11, 14, 25]. First, devices may adopt different implementations of RDMA engines. It happens among not only different vendors but also different generations of devices of the same vendor. We have investigated the impact of various devices in a large-scale storage system that involves two generations of Mellanox RNICs, which have different variants of DCQCN. In a hybrid deployment of 16 50Gbps CX-4 and CX-5 NICs, we observed a severe bandwidth imbalance, where the average throughput of CX-4 NICs degrades to 28Gbps over a full-mesh traffic pattern. Furthermore, we test the congestion control behaviors of NICs from different vendors. Specifically, Mellanox RNICs set the same congestion control rate for packets with the same destination IP, while Intel E810 RNICs enforce congestion control based on flows with the same five-tuple. In addition, Broadcom RNICs [7] implement DCTCP [2, 6] as the congestion control algorithm, while Intel E810 RNICs implement a window-based DCQCN variant [19]. The different congestion control algorithms can further amplify the bandwidth imbalance.

Second, RDMA requires Priority-based Flow Control (PFC) to maintain a lossless network fabric. Diverse devices increase the risk of generating PFC pause frames, which can propagate to the whole network and cause the network to stop forwarding traffic. In addition, the parameter tuning for the PFC configuration is time-consuming on newly deployed devices [25, 57], which usually takes weeks or months in large-scale networks with multiple vendors. During the long-term operation of production networks, we have observed multiple sources of PFC pause frame generation at both end-hosts and switches. Specifically, we found that implementation bugs

of switches and RNICs are one important root cause of PFC storms [11, 14]. In our datacenter, we record that the high loss rates occur due to diverse devices abnormality, system mis-configuration, and congestion of burst traffic in the production system, which has also been reported in prior work [58].

To cope with these issues, we need an open and unified framework to address the growing diversity of datacenter devices and give users the flexibility of RDMA programming to reduce the operational complexity of large-scale datacenter networks. Our key insight is that the RDMA data path, including memory semantics, needs fast and high-performance packet processing. In contrast, the control path including congestion control algorithms and the reliable re-transmission mechanisms, which are RTT-based operations, is relatively slow but needs to guarantee efficiency. This inspires us to rethink the functions division between hardware and software by on-loading congestion control and reliability modules to the software plane while strengthening the data path transport by following the standard RoCEv2 specifications [3–5] in the high-speed hardware.

We present Flor, an *open, unified* framework to support applications over heterogeneous networking devices. Flor *separates the data-path and control-path of RDMA transport with a hardware and software co-design* [24, 28, 37, 45, 52]. The data-path functions, *e.g.*, packet processing and bulk memory transfer semantics, remain on the hardware. Flor’s data-path follows RDMA primitives without any modifications in hardware to maintain high performance. Flor strengthens Reliable Connection (RC) transport through hardware/software co-design to overcome the low-efficient hardware-based Go-Back-N retransmission [14]. Furthermore, we leverage Unreliable Connection (UC) transport [4] as the first citizen for out-of-order demands in datacenters [42, 46] as it supports the out-of-order delivery of messages between RDMA operations without any requirements on the hardware change. We adopt UD as a key element to enable selective retransmission [36] for RoCEv2 and deliver messages to the applications in an out-of-order manner [42].

The control-path includes a load-aware dynamic chunking module, an RDMA-semantic-compatible reliability module, and a congestion control module. The load-aware dynamic chunking module balances between the performance and the software control granularity. Flor proposes a software selective retransmission scheme by leveraging UC to process *out-of-order delivery*. Flor implements an RTT-based congestion control algorithm similar to Swift [26] but improves the RTT measurement accuracy of previous work [28] by  $10\times$  on 99<sup>th</sup> percentile and 99.9<sup>th</sup> percentile RTT. By onloading these functions to hosts or programmable devices [40] (*e.g.*, IPU core [18], DPU core [51], CPU or even GPU [1]), the software developers have the flexibility of customizing and generalizing these functions across heterogeneous RNICs. For example, Flor can also adopt emerging congestion control schemes [26, 30] and optimization of transport protocols

(*e.g.* Swift, HPCC) instead of waiting for months or years of hardware upgrades.

We evaluate Flor through extensive experiments in an RPC benchmark and real production systems. We compare Flor with a customized RDMA library, XRDMA [32]. XRDMA implements the RPC interfaces with vanilla RDMA primitives. Compared to XRDMA, which suffers significant performance loss with 1/4096 packet loss ratio with lossy RoCE accelerations [53], Flor maintains steadily high throughput. Specifically, by deploying an RTT-based congestion control algorithm, Flor can handle 6000:1 incast with no throughput loss at run time. Flor achieves comparable performance as XRDMA for intra-pod communication and better performance than XRDMA for inter-pod communication. Our evaluations show that Flor reduces the bandwidth gap from 21.8% to 1.3% in the hybrid CX-4 and CX-5 deployment clusters and mitigates the performance gaps by 220% for RNICs of different vendors. For the production systems running big-data applications and cloud storage service, compared with XRDMA, Flor improves the job completion time of a big-data application job by 10% and achieves comparable latency and IOPS on the latency-sensitive cloud storage service. Specifically, the process of upgrading the existing RDMA framework, *i.e.*, XRDMA, to Flor has little performance impact on the running applications. Our practical experience with Flor shows that Flor provides a non-stop and smooth upgrade from lossless RDMA to lossy Flor.

In summary, this paper makes the following contributions:

- The interoperability of devices in RDMA networks needs be better addressed. We study the impact of heterogeneous RNICs in the production network.
- We revitalize the RDMA support by introducing an open unified framework accommodating primary RNICs in the lossy datacenter networks.
- We implement the framework and verify its effectiveness and low software overhead in both testbed and realistic production systems.
- As far as we know, this is the first systematic work considering the operation with heterogeneous devices, which innovates future RDMA system design from the perspective of service providers.

## 2 Background & Motivation

### 2.1 RDMA Preliminaries

RDMA is a hardware transport that exposes network operation through verbs API. User-space applications initiate data transmission requests to RNICs by posting Work Queue Elements (WQEs) into queue pairs (QPs). After transmitting the data, the RNICs generate Completion Queue Elements (CQEs) into Completion Queues (CQs) as the transmit completion signals for users. RDMA supports three transport types: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD) [4]. Correspondingly, it

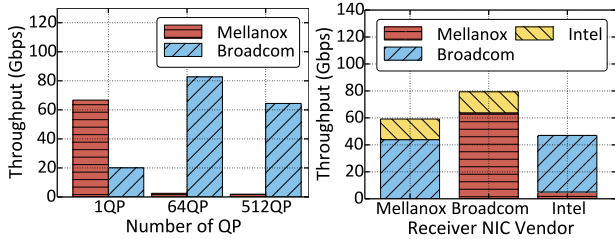
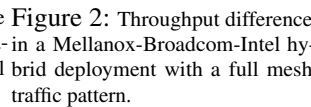


Figure 1: Throughput difference of a Mellanox RNIC and a Broadcom RNIC sending to an Intel RNIC in a Mellanox-Broadcom-Intel hybrid deployment with a full mesh traffic pattern.



NIC when the number of connections increases, *i.e.*, 64 QPs and 512 QPs. This causes unfair bandwidth share between applications hosted atop different RNICs. Figure 2 shows the throughput of each NIC with a full mesh traffic test (*i.e.*, all-to-all traffic) among the three NICs. We stack the throughput of each RNIC to the same receiver RNIC. For example, when a Broadcom NIC and an Intel NIC send traffic to a Mellanox NIC, the Broadcom NIC and the Intel NIC get the throughput of 43Gbps and 15Gbps, respectively (left bar). We can observe a similar performance variation when any two of the RNICs are competing with each other.

provides two well-known primitives: *SEND/RECV* are two-sided operations supported by all transports. *WRITE* is a one-sided operation supported by RC and UC, but *READ* is only supported by RC.

The difference in RNICs throughput is significant and leads to the computing tasks load imbalance on the nodes. We find that the root cause is the congestion control implementation difference or the congestion control algorithm difference among these heterogeneous RNICs. After we apply a unified congestion control algorithm, the performance gaps are eliminated (§7.5).

For connection-oriented services (UC and RC), QPs maintain Packet Sequence Number (PSN) and expected Packet Sequence Number (ePSN), respectively, on the sender and the receiver. RC QPs only receive packets with PSN correctly matching ePSN and increase ePSN after successful receiving. For RC, RNIC is responsible for retransmissions, where the receiver RNICs send acknowledge (ACK) packets. Once a packet is dropped, the sender must start retransmitting the lost packet. For UC, the transport does not retry messages with errors, and users must handle the error. Specifically, for UC QPs, when a packet of a verb is lost, the RNICs drop the whole verb, update ePSN, and continue to receive other verbs. Thus, compared with RC, it is more flexible for users to deal with out-of-order messages and potentially provide effective upper-level RPC service through software-defined reliability.

**Operational challenges caused by PFC storming.** PFC storm is a well-known problem [14, 15, 54, 57] that threatens the system’s availability if the pause frames are sent to the whole cluster [14]. RDMA systems in production adopt multiple mechanisms to mitigate the impact of these risks, such as PFC monitoring and watchdog, limiting the scale of PFC in a pod.

## 2.2 Production Experience

However, the PFC risk is not thoroughly eliminated and happens repeatedly with new causes. In addition to the known reasons of PFC storms, *e.g.*, the *slow receiver* [14] and switch hardware bug [11], we found that Machine Check Errors (MCE) caused by memory Error Correcting Code (ECC) and the lack of memory bandwidth can lead to the PFC storm when we introduce new RNICs in the datacenter. When MCE occurs on a server, RNIC receives data but can not DMA the data to the server memory. Thus, it sends excessive PFC pause frames to the neighbour switches and then spreads to the network. The occurrence frequency of MEC can be up to 1% [34], which leads to operational difficulty.

We present our production experiences demonstrating the difficulties of deploying the RDMA NICs of various generations and vendors in the same datacenter.

**Interoperability of heterogeneous RNICs.** We first investigate the performance gap between RNICs of different generations belonging to the same vendor. We run IB Perftest<sup>1</sup> in a cluster where 8 servers are equipped with CX-4 and another 8 servers are equipped with CX-5 RNICs. Given a full-mesh traffic pattern, *i.e.*, all servers send requests to each other, the throughput of CX-4 and CX-5 is 28Gbps and 41Gbps respectively. The throughput gap is 13Gbps (46.4%).

The lack of memory bandwidth also leads to PFC storms because CPUs and RNICs share the memory bandwidth on a server. When the CPU running applications preempts too much memory bandwidth, the memory bandwidth left for the RNIC is less than the network bandwidth [41]. Then the RNICs send PFC frames to prevent packet loss due to the RNIC buffer overflow. It is difficult to guard against every possible cause of the PFC storm. We expect to eliminate PFC from our production system while achieving performance compatible with a lossless network.

We then investigate the performance gaps among RNICs of different vendors. As shown in Figure 1, when a Mellanox RNIC [51] and a Broadcom RNIC [7] send traffic to an Intel RNIC [17] simultaneously. The configurations of the RNICs are depicted in §7.5. The Mellanox NIC gets 66Gbps, and the Broadcom NIC gets 20Gbps (220% of the performance gap) when each initiates one connection, *i.e.*, 1 QP. The Mellanox NIC gains less bandwidth, *e.g.*, 3Gbps, than the Broadcom

## 2.3 Motivation

These practical issues prompt us to rethink the usage of RDMA from the perspective of service providers. We aim to design an open and unified RDMA framework, which meets the following objectives:

- **Compatibility.** The open framework needs to be backward compatible with the legacy devices configured in the cluster.

<sup>1</sup>IB Perftest is a benchmark tool for measuring the throughput and latency of RDMA operations [13].



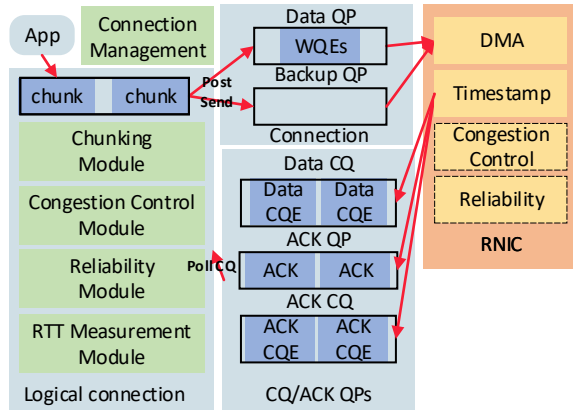


Figure 3: The framework of Flor

To this end, we abstract common features of available RNICs of main vendors and simplify the design by using the minimum set of functionalities in the hardware, *e.g.*, packetization, packet processing, message assembling, etc.

- **Flexibility.** To meet various application requirements and dynamic deployment of disaggregated services, the framework should support high feature velocity. It is programmable to realize efficient user-defined control mechanisms like new congestion control.
- **Availability.** The modern large-scale datacenters are built on Ethernet, which is a lossy network with multiple paths. Our framework is able to mitigate the impact of lossy Ethernet but fully utilizes the available network resources and maintains high performance [34, 42, 46].

## 3 Flor Design

### 3.1 Design Rationale

By investigating the RNICs of primary vendors, we notice that they follow the same RoCEv2 specifications in data-path but develop vendor-specific (even version-specific) control-path. Our key insight is that the RDMA data-path should be stable by following the standard specifications. At the same time, control-path needs to guarantee flexibility and availability. We can onload a subset of the transport functions to the software to provide programmability to developers. The design rationale is

#### Maintaining RDMA data-path specifications in hardware layer.

RDMA data-path, including packet processing and memory semantics, is a per-packet-based operation which is fast and high-performance. To maintain low latency and low CPU utilization features of RDMA, Flor places the data-path in hardware, which covers packetization, packet processing, message assembling, and direct memory access between RNICs and host memory. Therefore, Flor is compatible with primary RNICs.

**Onloading control-path to software layer.** The most flexible features in the control-path are congestion control and transmission reliability with regard to the lossy Ethernet.

The corresponding algorithms rely on the signals of packet latency, ECN notification, Inband Network Telemetry (INT), etc., the response interval of which can be several RTTs. Thus, Flor can onload the relatively slow control to the software layer by leveraging its programmability but has little influence on system efficiency. Notably, with the development of programmable devices, we can realize functions of control-path in not only hosts but the NPU [18] of RNICs or DPUs [51].

### 3.2 Architecture

Flor is an open, unified, high performance RDMA framework over lossy Ethernet in large-scale datacenters. The architecture is shown in Figure 3.

**Data path.** Since the process details of RDMA operations on QPs are dictated in the RDMA protocol [4], utilizing standard RDMA operations, which are supported by all RNICs, to transfer data among heterogeneous RNICs can achieve comparable performance. Flor takes RDMA **WRITE** and **SEND** as the base WQEs for the data transferring and receiving because they both support RC and UC and maintain high performance of RDMA. Notice that RDMA **READ** has a known performance issue [21] and only supports RC. Flor uses the RDMA **SEND** WQEs to transfer small messages (*e.g.*,  $\leq 32\text{KB}$ ) and **WRITE** to transmit large messages. For large messages, Flor needs an extra round-trip to exchange remote memory address and buffer size with remote servers. Note that the memory information exchange requires once for each large message. Flor prioritizes the **SEND** WQEs over large messages to avoid that head-of-line blocking to the small messages. On top of the RDMA verbs, Flor provides a *message-based* communication interface to support RPCs favoured by most datacenter applications [20].

**Control path.** The control-path of Flor consists of five flexible software modules: *Connection Management*, *Chunking*, *Reliability*, *Congestion Control*, and *RTT Measurement*.

- *Connection Management.* This module establishes and releases connections and manages backup QPs. Data are transmitted through QP and backup QPs, which take over the RDMA requests in place of the malfunctioned primary QP. The data CQs provide data completion events. The ACK QPs and CQs are used for sending and receiving software ACKs when using software reliability. Through QP management, Flor can abstract these backup QPs and present them as one QP to upper-level applications.
- *Chunking.* The *Chunking* module splits large messages into small RDMA requests, *i.e.*, *chunk*. Flor takes the *chunk* as the base unit of the selective repeat algorithm and congestion control algorithm, instead of a packet at the traditional transport [26, 57]. A chunk is sent to the network via a **WRITE** or **SEND** Work Queue Element (WQE).
- *RTT Measurement.* The *RTT Measurement* module collects the NIC hardware timestamp, synchronizes the hardware and software timestamp, and then updates RTT, which is

used as the signal of network congestion, retransmission and link failure detection.

- **Reliability.** Each WQE is passed to the *Reliability* module, which stores the transport information and maintains the state for packet loss detection and RTT calculation. We develop a software selective retransmission mechanism by tracking every request sent to the network.
- **Congestion Control.** The *Congestion Control* module takes the congestion signals, e.g., RTT, ECN, INT [39], and drop events, to calculate the congestion window or sending rate according to the congestion control algorithms. By default, we apply an RTT-based congestion control algorithm to eliminate the impact of complicated parameter tuning of congestion control on diverse switches [26].

### 3.3 Optimization and Deployment.

The key challenge for Flor is to offer flexibility while providing comparable performance to the vanilla RoCEv2 stack. We make the following optimizations:

- **Maintaining RDMA performance using the software/hardware co-design.** The RDMA hardware solution provides high throughput, low latency, and low CPU overhead. To maintain these advanced properties, we adopt a dynamic chunking mechanism to tune the size of messages for slow control-path when tracking the software congestion control and loss recovery. The overhead of the control-path functions onloading to the software layer is low because we apply large granularity of messages instead of packet processing (§ 4).
- **Enhanced UC with Selective Retransmission.** The packet loss rate in datacenters is actually low, which will not trigger frequent re-transmission. Designing a correct reliability mechanism while keeping the zero-copy memory semantic is the main challenge that Flor handles. UC has the property that RNICs can deliver the messages to the host without waiting for the previous ones to complete. Flor leverages this property to design a more efficient retransmission scheme, i.e., selective retransmission [36] without any hardware change to speed up the application processing [42] (§ 5).
- **Enhanced RC with Correctness.** RC is one of the data-path transport supported by Flor. Go-back-N, the RC's retransmission mechanism, is known to have low efficiency [36]. Flor enhances the Go-back-N mechanism by adding an additional software retransmission scheme. We address the correctness issue introduced by the software retransmission, where the retransmitted RDMA operators may overwrite the memory region that has been submitted to applications (§ 6).

Flor users can select a combination of these software modules in different scenarios. The software congestion control and reliability modules can be bypassed or replaced by the hardware functionalities. Table 1 shows some recommended configuration combinations for different deployment

Scenarios	Chunking	Reliability	CC
Intra-pod, PFC-enabled	No	RC	HW
Intra-pod, PFC- and ECN-disabled	Yes	RC or UC	SW
Across-pod Applications	Yes	UC	SW
CX-4/5 Hybrid	Yes	RC or UC	SW

Table 1: Recommended choices of modules in some scenarios. CC represents congestion control. HW indicates hardware-offloaded modules, SW indicates software-implemented modules.

scenarios in our production. For example, in a PFC- and ECN-disabled CX-4 cluster, Flor provides RDMA service by enabling chunking and software congestion control with hardware-based (RC) or software-based (UC) reliable transport. For cross-pod applications, software-based (UC) reliability is recommended to tolerant packet loss.

## 4 Dynamic Chunking

The chunking algorithm determines the granularity of RDMA requests bursting into the network. A large chunk size may result in a traffic burst that causes congestion and incurs high recovery costs in case of packet loss, while a small one leads to more CPU costs. Therefore, the key design point is dynamically adapting chunking size according to the current network status, which helps achieve both good performance and fine-grained traffic control. More specifically, it tries to adopt large chunk sizes with hardware SEND or WRITE operations to reduce CPU cost when the loss rate is low. Once packet loss occurs, it applies chunk-slicing and retransmission of dropped chunks by software.

Flor uses the estimated RTT as the default feedback signal of network status for the dynamic chunking strategy. The estimated RTT is not only used as the feedback signal of the dynamic chunking algorithm but also used as the congestion control signal, as well as a timeout signal for reliability.

### 4.1 Accurate RTT Measurement

The accuracy of RTT measurement directly impacts the performance of all these components. Different from the approach [26] where measures RTT based on per-packet timestamp, Flor measures RTT for the chunks with different sizes. RoGUE [28] firstly devises a way of RTT measurement for dynamic verb sizes on RNICs and utilizes the hardware timestamp functionality of RNICs to get an accurate timestamp to calculate RTT. Flor takes the RoGUE's methodology to measure RTT for the RC transport and further improves the RTT measurement accuracy for UC transport.

Figure 4 shows the RTT measurement method in Flor for the UC transport. Note that leveraging hardware timestamp to measure RTT requires synchronizing the software and hardware clock (Appendix A.2.1). On the sender side, the timestamps of the data WQE sending completion ( $T_1$ ) and the corresponding ACK receiving completion ( $T_4$ ) can be obtained from the hardware. On the receiver side, the timestamp of the data WQE arrival ( $T_2$ ) is read from the hardware and the timestamp of the corresponding ACK WQE

posting time ( $T'_3$ ) is accessible through software. ( $T'_3 - T_2$ ) is computed in the receiver and sent back to the sender by a subsequent ACK packet. Thus RTT can be calculated with:

$$RTT = (T_4 - T_1) - (T'_3 - T_2). \quad (1)$$

However, the measurement is not absolutely accurate as  $T'_3$  is the post time rather than the actual send completion time of the ACK ( $T_3$ ) due to the queuing of sending requests in NICs. In addition, under a heavy load, the ACK can be delayed up to milliseconds upon receiving by the sender software due to the head-of-line blocking by the data WQEs and the QP scheduling policies. As a result, on the one hand, the measured RTT can be increased by this overhead, which inaccurately reflects the network congestion; On the other hand, this also prolongs the congestion feedback loop such that the sender can not respond to the network congestion in time.

Flor uses two optimization approaches to improve RTT measurement accuracy and shorten the feedback loop delay. First, Flor uses a high-priority UD QP to send and receive ACKs to avoid head-of-line blocking and scheduling delays on RNICs. Second, Flor uses a separate completion queue for ACKs and polls it prior to the data completion queue in each batch. Our measurement shows that the measured tail RTT without optimization can be up to several milliseconds due to QP scheduling (Figure 16 in Appendix A.2.2). The overestimated RTT can cause an unnecessary window decrease. Using a separate QP and completion queue for ACKs improves the RTT measurement accuracy by  $10\times$  on tail RTTs.

## 4.2 Chunking Strategy

Flor extracts the chunking algorithm as a module such that users can specify their own chunking algorithms. Here we present the default algorithm used by Flor. The key idea is to dynamically reduce the chunk size when the RTT of network gets worse and increase the chunk size when its status gets better. We initialize the chunk size by the minimum value of the available congestion window ( $acwnd$ ) or bandwidth-delay product (BDP) ( $chunk\_size \leftarrow \min\{acwnd, BDP\}$ ). Then we update the chunk size for each RTT.

As shown in Algorithm 1, we use estimated RTT, which reflects network queuing, as the feedback signal of chunking in Flor. We maintain two smoothed RTTs ( $rtt_s$  and  $rtt_l$ ) with Exponentially Weighted Moving Average (EWMA) [31] using different indexes  $\alpha$  (the parameter that controls the weight of new feedback). The short-term RTT  $rtt_s$  demonstrates the up-to-date congestion status, while the long-term RTT  $rtt_l$  indicates the common status of the connection.

Generally, we define a span of expected RTT denoted by  $(\beta_{max} * rtt_l - \beta_{min} * rtt_l)$ . Here  $\beta_{max}$  and  $\beta_{min}$  are configurable parameters to identify the upper and lower bounds.  $(\beta_{max} * rtt_l - rtt_s)$  indicates the position of short-term RTT in the RTT span. The Algorithm divides the RTT span linearly, as shown in line 4 of Algorithm 1.  $size_{max}$  is the maximal chunk size

<sup>2</sup> For example, if the  $rtt_s$  reduces to the lower bound of RTT

<sup>2</sup>By default,  $size_{max}$  is 64KB, which is a trade-off between the CPU

span ( $\beta_{min} * rtt_l$ ), it indicates that the status of network is good and the chunk size increases to the  $size_{max}$ . On the contrary, if the  $rtt_s$  increases to the upper bound of RTT span ( $\beta_{max} * rtt_l$ ), it indicates that the load of the network is high and we should reduce the chunk size to the  $UNIT\_SIZE$ .

---

### Algorithm 1 An RTT-based Chunking Algorithm

---

**Input:** RTT  $rtt$ , available congestion window  $acwnd$

**Output:**  $chunk\_size$

- 1: update short-term RTT  $rtt_s$  with  $rtt$ ,  $\alpha_s$  via EWMA
  - 2: update long-term RTT  $rtt_l$  with  $rtt$ ,  $\alpha_l$  via EWMA
  - 3:  $size_t \leftarrow size_{max} * (\frac{\beta_{max} * rtt_l - rtt_s}{\beta_{max} * rtt_l - \beta_{min} * rtt_l})$
  - 4:  $chunk\_size \leftarrow \min\{size_t, acwnd\}$
  - 5: **return**  $chunk\_size$
- 

Note that the chunking module still applies to have a fine-grained traffic control if Flor uses a rate-base congestion control, e.g., DCQCN. To support the reliability design (§3.2), Flor aligns the  $chunk\_size$  to  $UNIT\_SIZE$  (the minimal  $chunk\_size$ ), i.e.,  $chunk\_size$  is exactly multiple times of  $UNIT\_SIZE$ . Note that to prevent deadlock of the congestion window, the  $chunk\_size$  is set to  $UNIT\_SIZE$  when the available congestion window is smaller than  $UNIT\_SIZE$ . The  $UNIT\_SIZE$  can be equal to the value of the current MTU. To mitigate the impact of packet loss, Flor adopts different chunking mechanisms for RC or UC transport, respectively. For RC transport, Flor directly reduces the large chunk sizes to the minimum chunk size of one  $UNIT\_SIZE$  to reduce retransmission overhead and avoid the live lock of retransmission. For UC transport, Flor relies on congestion window, which will cut its size by a half upon a packet loss. And finally, the chunk size becomes  $UNIT\_SIZE$  when continuous packet loss happens. In particular, Flor can adopt Selective Retransmission with UC in Section 5 to reduce chunk retransmission under high packet loss and achieve better transmission efficiency than Go-Back-N.

## 5 Selective Retransmission with UC

UC transport only drops the verbs that have packets dropped but does not drop the subsequent successfully-delivered verbs. Thus, based on the chunking mechanism that splits RDMA messages into varied sizes of WQEs, Flor is able to design reliability mechanisms (sequence number, acknowledgement, and retransmission) at the granularity of chunking WQEs. The transmission of RDMA *WRITE* operations does not need any reordering buffer because *WRITE*s are directly DMA-ed into the host buffer. However, there are still some challenges to implement reliability in software for one-sided RDMA *WRITE* operations:

- **Challenge #1: Additional data copy.** Since RDMA *WRITE* writes an array of memory pieces to one continuous

efficiency to transmit large chunks and the retransmission overhead of packet loss to transmit smaller ones.



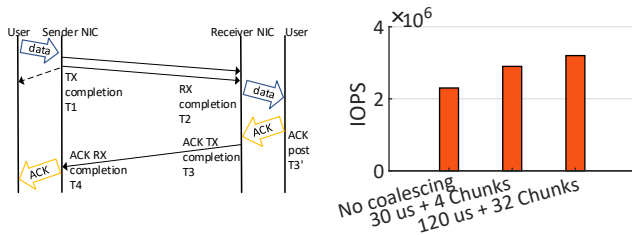


Figure 4: RTT measurement for UC in Flor.  $T1$ – $T4$  are timestamps of events.

remote address, Any extra content added to the chunks of a large message, including information needed by reliability mechanisms such as sequence numbers, can break the original message, which then incurs a memory copy at software to assemble the scattered memory into the original message.

- **Challenge #2: Software ACK.** *WRITE* is a one-sided RDMA operation that bypasses the CPU of the receiver. The receiver can not know whether a *WRITE* succeeds or fails, which is one of the main issues when the one-sided operator is used in practice. Flor uses WQEs to encode software ACK messages as UC does not generate hardware ACK packets. Without careful design of the ACK message, the high frequency of the software ACK will significantly degrade the performance.
- **Challenge #3: Repetitive memory access.** The retransmission of RDMA *WRITE* operation may cause a data integrity issue as RNICs can write to a piece of memory already submitted to the application.

To solve these problems, some novel designs, including sequence number space for *WRITE\_WITH\_IMM*, software ACKs, and two-sided retransmission, are adopted in Flor’s reliability mechanism.

**Sequence number space for *WRITE\_WITH\_IMM*.** To assemble the chunks into the original message on the receiver without the extra data copy (**Challenge #1**), Flor uses *WRITE\_WITH\_IMM* to generate signals to software for the arrival of chunks. One feature of *WRITE\_WITH\_IMM* is that it can carry an extra 32-bit *imm\_data* set by the sender. With *WRITE\_WITH\_IMM*, the receiver can detect the arrival of RDMA verbs and receive the chunk number without polluting the application memory. For *SEND*-transported small requests, Flor adds an additional header in the payload to carry additional information, including the sequence numbers.

However, to reassemble the initial messages from the chunks, we need another sequence number for the chunks. Note that chunks of a large message are also not continuous in the sending queue since some *SENDS* operations (*e.g.*, retransmissions and the address-exchanging messages of *WRITE*) are prioritized in Flor.

Similar to QUIC [27], Flor encodes two sets of sequence numbers into each RDMA WQE: *global sequence number*

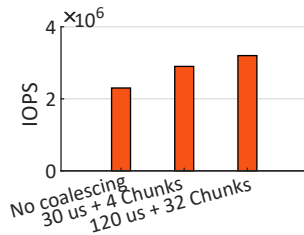


Figure 5: Performance impact of ACK coalescing.

and *reliability sequence number*. The global sequence number is a 64-bit value, and all the RDMA WQEs have a unique global sequence number to identify the sequence within a QP. The reliable sequence number is used to identify the sequence within the same type of WQEs, *i.e.*, *WRITE* WQEs and *SEND* WQEs have separate reliable sequence number space. These two sequence number spaces also allow Flor to identify the original WQEs and the retransmitted WQEs by different global numbers such that ACK information and the timestamp carried in ACKs are clear. The retransmission WQEs share the same reliable sequence number with the original WQE. More details can be found in Appendix A.1.1. **Software ACK.** Flor acknowledges every WQE, but generating an ACK for each WQE can cause high overhead for small-message traffic (**Challenge #2**). Flor puts multiple sequence numbers into one software ACK to reduce the CPU overhead. The detailed ACK format is explained in Appendix A.1.2. However, ACKs should be sent timely since ACKs carry RTT and WQE numbers used in the congestion control and reliability mechanism. Thus, Flor sets these trigger rules for receivers to send an ACK immediately: (1) the cumulative number of WQEs; (2) the cumulative size of WQEs; (3) the last WQE in the congestion window signed by a hint bit in the header or *IMM\_DATA* field; and (4) an out-of-order reliable sequence number, whichever reaches first. Flor also sets a timer as a trigger because the tail of a flow or small bursts may not have enough data to trigger the cumulative counters.

To show the impact of different ACK triggering frequencies, we set up an experiment with a client and a server, and each is equipped with a CX-4 dual-port NIC. There are 8 threads and 64 QPs on each node. Figure 5 shows the IOPS in a 128-byte request and response RPC benchmark with different ACK triggering mechanisms. The ACK coalescing mechanism improves ~40% IOPS compared to a per-WQE ACK mechanism (*No coalescing* in Figure 5). In addition, among different coalescing WQE sizes and timers, the setting of 120 $\mu$ s timer and cumulative counter of 32 WQEs achieves a satisfying IOPS, which is the default configuration of Flor.

**Two-sided retransmission.** When the sender detects an out-of-order delivery from an ACK or a timeout event, it retransmits the WQEs. Flor handles the retransmissions by *SEND*, since spurious retransmissions of *WRITE* may cause a data integrity issue (**Challenge #3**). The data integrity issue can happen when WQEs are queued in the network for a long time, which incurs timeout retransmission at the sender. In this case, the original WQE is received by the receiver, and the whole message is submitted to the upper-layer application. The application can write the content of the message as it needs. However, the subsequent retransmitted *WRITE* WQE arrives at the receiver and overwrites the memory region that the application has already changed without informing the CPU of the receiver. Flor retransmits WQEs by *SEND* through the same QP for the lost one to avoid uncontrolled



memory access from RNICs. In a *SEND* operation, the data is written into a piece of pre-posted memory. Flor then copies the data into its desired location if the message has not been submitted to the application. If the retransmission *SEND* arrives earlier on the receiver, the following original *WRITE\_WITH\_IMM* will be dropped by the receiver RNIC because its hardware packet sequence number is smaller than the expected hardware packet sequence number, which is updated because of the arrival of the *SEND*.

## 6 Enhance Hardware Retransmission

Flor is compatible with hardware reliability by using RC to reduce software costs and achieve better performance. Note that Flor supports both one-sided and two-sided RDMA operations on RC QPs. The hardware retransmission of RDMA operations is out of the control of software congestion control. This has potential risks of incurring network congestion since the size of inflight data can be much larger than the software congestion window. Flor improves the hardware reliability by adding a software retransmission scheme. Flor sets short retransmission retry times in RC QPs to limit the size of data exceeding the software congestion control window.

If the retransmission fails for the retry times, the QP is turned into error states by the RNIC hardware. Turning the error states of the QPs into the working states takes a long time, e.g., 5ms. Instead, Flor resubmits the uncompleted inflight WQEs to another pre-connected QPs, called backup QPs [28], and flips the backup QP to be the primary QP to continue data transmission. At the same time, Flor re-connects the original QP in the background. The inactive backup QPs do not consume additional cache resources on RNICs and thus have no side effect on performance [28]. Our practical experience shows that using one retry time incurs too many QP switches in some extreme cases e.g., large-scale incast. By default, Flor uses two retry times and two backup QPs for each connection. Compared to QP reconnecting, switching to backup QPs costs less time, i.e.,  $\sim 60\mu\text{s}$ .

A racing issue occurs when QPs turn into the error state occasionally: the sender may return a failure of a WQE while the receiver successfully receives it. This case happens when the QP at the sender turns into the error state with successful inflight operations. In such a case, the sender posts a duplicated WQE mistakenly on the backup QP of the logical connection. Another corner case that causes the same problem is that the sender times out when the hardware ACK is on the flight. Flor retransmits the WQEs with RDMA *SEND* and checks if the message has been submitted to the application before the data are copied into the destination application's memory.

## 7 Evaluation

We evaluate Flor by answering the following questions:

1. The software overhead of Flor in the 100Gbps network (§7.2) and its robustness against packet loss (§7.3)?

Cluster	Nodes	RNICs
A	100	CX-4 Lx 25Gbps dual-port
B	16	8 × CX-5 25Gbps dual-port 8 × CX-4 Lx 25Gbps dual-port
C	48	CX-5 100Gbps dual-port
D	3	Intel E810 100Gbps dual-port Broadcom P2100G 100Gbps dual-port Mellanox BlueField-2 100Gbps dual-port

Table 2: Clusters setups used in our evaluation.

2. The behaviour of Flor in both intra-pod and inter-pod communications when PFC is disabled (§7.4)?
3. The effectiveness of Flor in a hybrid deployment with heterogeneous RNICs (§7.5)?
4. The performance of Flor's default *Congestion Control* in large-scale incast scenario (§7.6)?
5. The impact on services when upgrading from the current network to Flor in production systems (§7.7)?

### 7.1 Experiment Setup and Benchmarks

**Cluster setup.** Table 2 lists four clusters used in the evaluation. The default RDMA configurations of our clusters are that: (1) for CX-4 lossless RNICs, PFC is enabled on ToR and Leaf switches but disabled on Spine switches; (2) for CX-5 lossy RNICs, PFC is disabled on all switches, and the *lossy RoCE acceleration features* [53] are enabled.

**Baseline and workload.** Our RPC system used in evaluation supports XRDMA, Flor, user-space TCP, and kernel TCP. XRDMA is a vanilla RoCEv2-based RPC library, which is deployed in the clusters listed in Table 2 before upgrading to Flor. The user-space TCP is based on DPDK. Two applications run on top of our RPC framework: a Map-Reduce-like application and a distributed block storage service.

**Configuration for Flor.** In the clusters, we configure one specific priority queue on both switches and RNICs. The PFC and ECN are enabled on this priority queue (lossless queue). Besides, we reserve another priority queue (lossy queue) for Flor in which PFC and DCQCN are disabled. The coalescing ACK parameters are 120 $\mu\text{s}$  timer, 32 WQEs and 32KB data at most. The base RTT used in the software congestion control is 50 $\mu\text{s}$ . The minimal and maximal chunk size is 1KB and 64KB, respectively.

### 7.2 Software Overhead.

Flor introduces additional CPU cost due to the software implementation of reliability mechanism and chunking. To evaluate its impact, we compare the single-core performance of different network protocol stacks with our RPC benchmark, including XRDMA, Flor (RC), Flor (UC), user-space TCP, and kernel TCP. The I/O depth (i.e., the maximal number of inflight RPC requests) is 8. We vary the RPC request size from 4KB to 1MB and fix the response size at 128 bytes. The servers are equipped with Intel CPUs (2.9GHz) which have 96 logic cores, and CX-6DX 100Gbps dual-port RNICs (not listed in Table 2).

Figure 6(a) demonstrates the throughput of all the stacks with different RPC sizes. The single-thread throughput of Flor

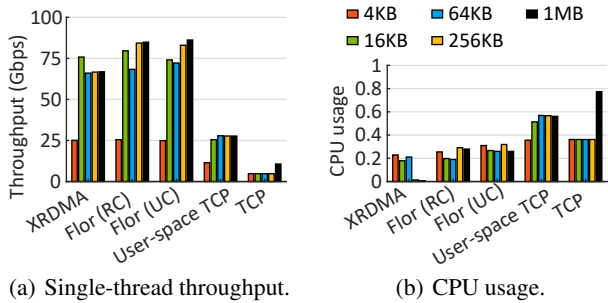


Figure 6: Single-thread throughput and CPU usage of Flor and other different network stacks.

and XRDMA can reach to  $\sim 88$ Gbps while user-space TCP and kernel TCP achieve up to  $\sim 30$ Gbps even with large RPC size, i.e., 1MB. Notice that due to the hardware and RoCEv2 protocol overhead, such as headers of Ethernet, IP, UDP, and IB with 1024B MTU, the standard RDMA benchmark, i.e., Perfest [13] achieves a maximum bandwidth of 88Gbps. This shows that the *Chunking* mechanism does not hurt the single-thread throughput of Flor. We observed that the performance degradation of Flor (RC/UC) happens at 64KB due to extra memory information exchanges for using RDMA WRITE to transmit each large message and dismisses as message sizes increase. The throughput of large requests in XRDMA is lower than Flor because XRDMA uses RDMA READ operation to transfer the large requests, and RDMA READ operation has inflight bound on RNICs along with some well-known performance issues [16, 23]. When using a chunk of 4KB, Flor can handle over 770K chunks per second with a single thread. Flor is able to maintain high throughput as the chunk size is usually larger than 4KB, and applications often adopt multiple threads.

We then estimate the corresponding CPU usage. The CPU usage is obtained from *perf* [38] tool since we use polling mode for RDMA. Notice that in our production storage system, it adopts a run-to-completion model based on a co-routine IO framework [11], where the network polling for disk read or write uses the same core with storage protocol processing in concurrent execution. As shown in Figure 6(b), Flor takes less than 0.3 CPU core for large data size of 1MB message in 100 Gbps network. Most modern servers usually have large numbers of cores ( $>96$ ) and 0.3 CPU cores usage ( $<0.4\%$  usage) has little impact on the production system.

Flor maintains low CPU cost because it leverages zero-copy features of RDMA and mainly deals with lightweight control events for congestion control and reliability. In addition, Flor is compatible with different platforms, which can further reduce the host CPU cost by offloading Flor to SmartNICs, and FPGA in computation-intensive hosts.

We also measure the single-core throughput of Flor at 200Gbps RNICs and show that Flor can achieve comparable throughput as vanilla RDMA. Besides, we show that Flor outperforms the other network stacks, i.e., SNAP [33], eRPC [20]

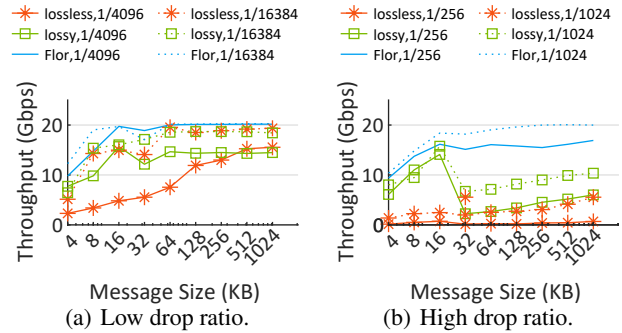


Figure 7: The throughput of lossless, lossy XRDMA and Flor under different packet drop ratio.

(as shown in Table 4).

### 7.3 Performance with Packet Loss.

To validate the effectiveness of the software *Reliability* mechanism of Flor, we take two CX-5 RNICs from cluster B in Table 2 and disable one port on each RNIC. We manually configure packet drop ratios on the RNIC of the receiver. We compare Flor against XRDMA using the lossless and lossy configuration of CX-5. The congestion control is disabled to avoid transmission rate back-off due to packet loss.

We use various request sizes (4KB~1MB) under four packet drop ratios (two high ratios of 1/256 and 1/1024, and two low ratios of 1/4096 and 1/16384). As shown in Figure 7, Flor outperforms the lossy and lossless setup across all the drop ratios. Due to software selective retransmission, Flor is able to maintain a performance close to that of zero packet drop at the low packet drop ratios. Its performance decreases slower than the lossy and lossless setup at the high drop ratios. We observe that the lossy acceleration feature achieves higher throughput with the occurrence of packet loss compared to the lossless setup, i.e., the lossy acceleration features of CX-5 does improve the packet loss recovery performance. However, the throughput of lossless and lossy RDMA both decrease dramatically when the drop ratio is larger than 1/4096. It indicated that the lossy acceleration features of current hardwares still cannot maintain good performance under high packet loss. Flor achieves similar results by performing the same experiments through manually configuring the random packet drop ratio on the port of the ToR switch connecting to the live port of the RNIC at the host.

### 7.4 Intra- and Inter-Pod Traffic

Flor uses an advanced congestion control to enable lossy RDMA support, eliminating the PFC dependency while maintaining high performance. We validate the performance gain of Flor in large-scale intra- and inter- pod transmission through cluster A (pod1) and B (pod2) (Table 2). We use the default configuration in our clusters in the tests of XRDMA: PFC for RoCE traffic is only enabled on ToR and Leaf switches (i.e., intra-pod) and disabled on Spine switches

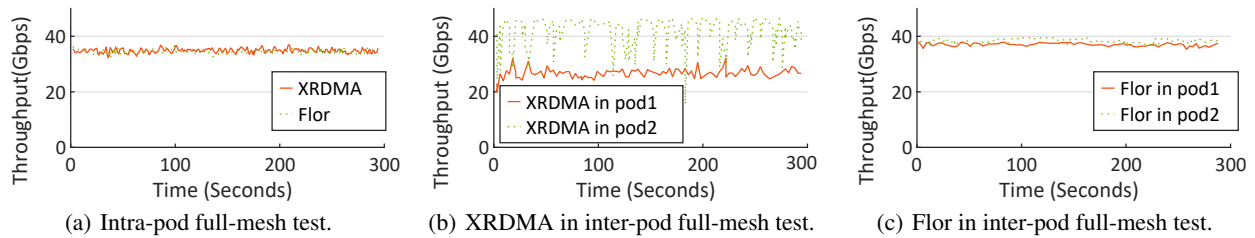


Figure 8: The throughput of XRDMA and Flor in cross-pod scenarios.

(i.e., inter-pod). Similarly, we configure the intra-pod traffic of Flor (UC) on the lossless queue and the inter-pod traffic of Flor (UC) on the lossy queue. For each node, a client sends large RPC requests (512KB) to each server on other nodes with IO depth of 8. The size of the RPC response is 128 bytes. Each client and server uses 4 threads in the tests.

As shown in Figure 8(a), for intra-pod traffic, the average throughput of Flor (UC) is comparable to XRDMA which is stable at  $\sim 35$  Gbps. For inter-pod traffic test, as shown in Figure 8(b) and 8(c), the throughput of XRDMA in one pod shakes fiercely between 35 Gbps and 45 Gbps, and the throughput in the other pod oscillates between 35 Gbps and 20 Gbps. The unstable and unbalanced throughput of XRDMA is caused by packet loss as PFC is disabled in inter-pod switches, and DCQCN can not prevent packet loss. In contrast, the throughput of Flor (UC) is stable and balanced between the two pods. In summary, Flor can achieve higher and more stable throughput than XRDMA for lossy inter-pod tests. The clients in pod1 suffer more from throughput loss because they send more cross-pod traffic and experience more packet loss in this full-mesh traffic pattern.

## 7.5 Heterogeneous RNICs

To evaluate the effectiveness of Flor over heterogeneous RNICs, we test with 8 CX-4 and 8 CX-5 RNICs in cluster B (Table 2) with PFC and DCQCN enabled. We run the RPC benchmark with a full-mesh traffic pattern atop of Flor and XRDMA. When using XRDMA, the average throughput of CX-4 and CX-5 RNICs is 33.2 Gbps and 41.3 Gbps, respectively. The throughput gap between CX-4 and CX-5 is 8.1 Gbps (24.3%). When using Flor, the throughput of CX-4 and CX-5 RNICs with Flor is 37.1 Gbps and 36.6 Gbps, and the throughput gap is 0.5 Gbps (1.3%). This indicates that Flor eliminates the throughput gap between CX-4 and CX-5 RNICs by replacing the hardware congestion control with a unified software congestion control, which minimizes the performance difference introduced by the control path of heterogeneous hardware.

To verify the effectiveness of Flor over RNICs from different vendors, we run perfest among 100 Gbps RNICs, including Mellanox BlueField-2 [51], Intel E810-C RNIC [17] and Broadcom NetXtreme P2100G RNICs [7]. For congestion control, BlueField-2 only supports DCQCN, P2100 only supports DCTCP, and E810-C supports DCQCN, DCTCP and

Timely. To clarify the unmatched performance introduced by different congestion control algorithms, we choose to set DCQCN for BlueField-2 and E810-C, and DCTCP for P2100G with PFC and ECN enabled on RNICs and the connected switches. Each sender issues 512 QPs and sends traffic with 64KB data blocks. Four Flor configurations, i.e., *PFC with hardware congestion control (CC) and Flor*, *hardware CC and Flor*, *Flor*, and *Flor with fixed cwnd* are tested. *Flor with fixed cwnd* means the software congestion control algorithm has a fixed congestion window and does not change throughout the whole experiments. We set the fixed congestion window size as one bandwidth-delay product in this experiment.

Figure 9 shows the throughput of each NIC under the full-mesh traffic pattern. BlueField-2 and P2100G get the same bandwidth when they are competing with each other to send traffic to E810-C. Compare with Figure 2, with the configurations of PFC + hardware CC + Flor (1), hardware CC + Flor (2) and Flor (3), and we see that Intel NIC gets the same throughput, i.e., 18 Gbps, when competing with Mellanox NIC and Broadcom NIC. We see the performance gap between Intel NIC and Mellanox NIC when they send traffic to the Broadcom NIC, even if we only apply the Flor’s congestion control (3), where the Intel NIC gets 19 Gbps, and the Mellanox NIC gets 56 Gbps, 194% of the performance gap.

The reason is that Flor’s congestion control takes the RTT as the congestion signal. Intel NIC has a higher packet processing time, which causes the estimated RTT between the Intel NIC and the Broadcom NIC to be higher than the one between the Mellanox NIC and the Broadcom NIC. Thus, the RTT-based congestion control algorithm reduces window size in Intel NIC and results in lower throughput. This indicates Flor’s congestion control needs to be further improved by taking the NIC processing delay into account. If we fixed the congestion window, i.e., Flor with fixed cwnd (4), the bandwidth is less-skewed shared between the Mellanox NIC and the Intel NIC.

## 7.6 Large-scale Incast

We build a group of incast tests in cluster A (Table 2) to evaluate the performance of Flor’s congestion control. We measure the throughput and Out-of-Sequence (OOS) counter. We configure Flor running on the lossy queue, i.e., disabling



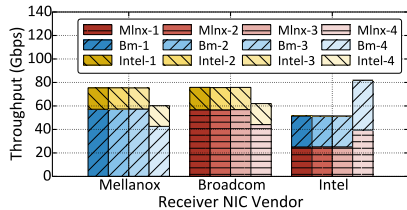


Figure 9: Bandwidth in a Mlnx-Broadcom-Intel hybrid deployment with Flor. Configurations: 1 = PFC+hardware CC+Flor, 2 = hardware CC+Flor, 3 = Flor, 4 = Flor +fixed cwnd.

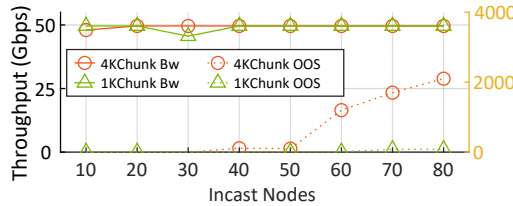


Figure 10: Flor performance in large-scale incast test.

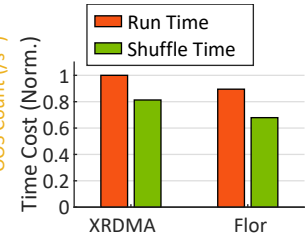


Figure 11: Run time in big data application.

PFC and DCQCN. For incast traffic, we install RPC clients on  $N$  machines and RPC server on one remote server. Each client has 8 threads connecting to the server, and each thread issues 32KB RPC requests with IO depth of 8. Thus, the number of QPs on each client is  $8 \times 8 = 64$ , and the number of QPs on the server is  $N \times 64$ . Thus, the maximum incast degree is  $\sim 6000$  when the node number is 90 in this test.

Figure 10 shows the throughput and OOS counters with different incast nodes of  $N$  and the minimal chunk sizes. Given the minimal chunk size of 4KB or 1KB, the throughput is consistent of 50Gbps with the increasing scale of incast. However, given the minimal chunk size of 4KB, the number of OOS increases when the incast nodes are larger than 50 (about 4000 : 1 incast). This is because the minimum chunk size of 4KB is the minimal congestion window size for each QP, and the volume of burst traffic is too large in such a large-scale incast. Therefore, we apply the minimal chunk size of 1KB, which avoids the generation of OOS in the large-scale incast. Notably, we set the minimal chunk size of 4KB in the storage production network because we have optimized the storage application to balance the load across nodes, which avoids such large incast events in practice [11].

## 7.7 Evaluation in Production Network

**Big-data applications.** ServiceX is a Map-Reduce-like big-data application that runs on top of the distributed storage service. The completion time of the shuffle processing influences the performance of the whole task, and it desires fast and stable network transmission. To estimate the impact of Flor in the production system, we run ServiceX atop of Flor and XRDMA in cluster C (Table 2) with a lossless network. We conduct the task of sorting 1TB of data. The number of mapper tasks and reducer tasks are both 1K, and each mapper processes data of 1GB. We apply two key metrics: the average running time of a mapper and the average shuffle time, *i.e.*, the time of transferring data in the network. As shown in Figure 11, in comparison with XRDMA, Flor reduces the average running time by 10% and accelerates the average shuffle time by 16% due to an efficient congestion control strategy.

**Non-stop upgrade.** Flor allows to upgrade online with negligible down time of service, which is crucial to meet service level requirements in modern datacenter. We measure the impact of upgrading from XRDMA to Flor on applications

such as Pangu [11] through the measurements of normalized throughput and latency and PFC counters. In the experiment, the software upgrading takes place at the 0.5<sup>th</sup> minute. As shown in Figure 12(a), the read and write throughput of the application have a slight jitter ( $< 2\%$ ) when switching to Flor. Figure 12(b) shows that the latency increases by 10% at 0.5 minute, lasting less than 30s. Figure 12(c) shows the generation of PFC pauses (packets per second, pps) and its duration time ( $\mu s$ ), which appears in a very short time period. At the 5.5<sup>th</sup> minute, we then disable DCQCN and PFC. The throughput is unaffected, and the latency decreases slightly ( $\sim 3\%$ ). This latency might be caused by the interference between DCQCN and the software congestion control. When running Flor with and without PFC and DCQCN, all the metrics are healthy.

**High-performance block storage service.** Flor is applied for latency-sensitive applications such as the Enhanced SSD (ESSD) product of Elastic Block Storage (EBS). ESSD provides block storage service (virtual disks) as local devices through high performance network. We compare the latency and requests per second (IOPS) of an EBS application running with XRDMA or Flor in cluster C (Table 2). We adopt the workload of a real ESSD storage application with an I/O size of 4KB. XRDMA is tested with RoCE lossy accelerations enabled. Flor is tested with these features disabled. There are three kinds of configurations for Flor: (i) Flor with hardware reliability and hardware congestion control (Flor HW R/C); (ii) Flor with hardware reliability and software congestion control (Flor HW R); and (iii) Flor with software reliability and congestion control (Flor SW).

Figure 13 (a) shows the normalized single-operation latency of XRDMA and Flor. Flor demonstrates comparable average latency performance with XRDMA among all the operation types. Although software-based modules are involved, the latency of Flor is still slightly lower (1%–8%) than XRDMA due to the optimized software stack of Flor. Flor (HW R/C) has the lowest average latency through hardware-based implementation. Flor (SW) and Flor (HW R) have slightly higher latency ( $< 3\%$ ) due to the overhead of software stack. Figure 13 (b) shows that Flor achieves the comparable normalized IOPS as XRDMA for 4KB *Read* and *Write*. In conclusion, in supporting block storage service, Flor has comparable latency and IOPS with XRDMA.



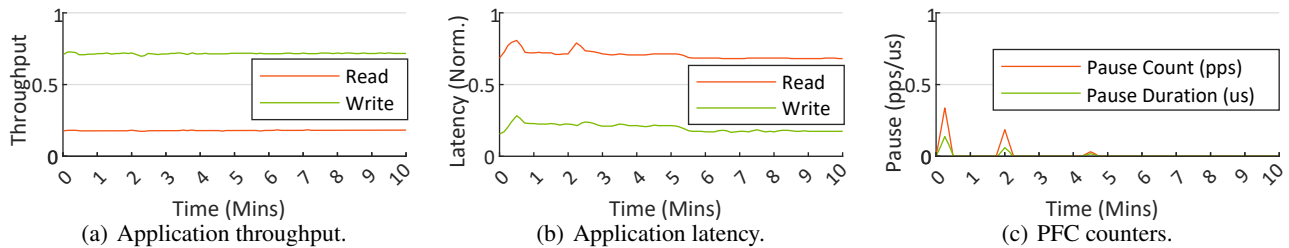


Figure 12: The system performance of upgrading from XRDMA to Flor.

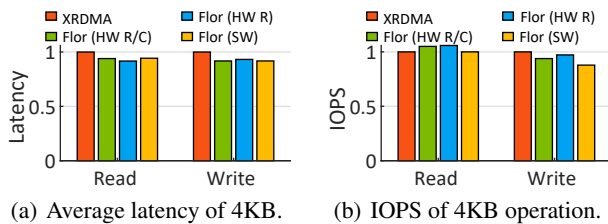


Figure 13: Performance of Block Storage Service.

## 8 Discussion

**Hybrid RDMA NICs’ deployment in datacenters.** In the production network with over 100K servers, new servers and new NICs are incrementally deployed, which results in the mix of the latest generation and earlier generations (or different vendors) of RDMA NICs co-existing in the same datacenter network. In addition, the malfunctioning servers in the built-up clusters can be replaced by servers configured with NICs of different generations or vendors. The new generation of RNICs is released by the vendors every 2 or 3 years. From our experience in Block Storage Service, we first deployed storage servers with CX-4 RDMA NICs in 2016. In 2019, we started to deploy new servers of CX-5 RDMA NICs because of higher performance and price ratio. As a result, there exists the hybrid deployment of both CX-4 and CX-5 RDMA NICs. The same deployment choice was made when we introduced CX-6DX RDMA NICs in 2021.

**Programmable control plane.** Flor provides a new perspective of layered architecture to achieve high velocity and performance with diverse hardware. Flor envisions the "programmability" of reliability and CC modules of RDMA in the control plane, which can be implemented in programmable hardware, such as smartNICs with embedded CPU or NPU, e.g., ConnectX-6, Bluefield, Intel IPU. In this way, we can make use of the programmable hardware capability such as hardware timestamp, rate-limiting, and packet drop detection to further improve the efficiency of CC and reliability while keeping the unified control policy among heterogeneous RNICs with the same Flor architecture.

**Concerns of implementation.** Currently, not all NICs support UC. First, this research work has demonstrated, for the first time, the effectiveness of using UC for high-performance out-of-order transmission in product networks. Second, UC is a standard transport defined in the RoCE specification, and its logic is simple and easy to implement by hardware.

Network Protocol	MTU (B)	Throughput 100/200 (Gbps)
Linux TCP	1500	22
SNAP	1500	39.1
SNAP (+I/OAT)	5000	67.5 (82.2)
eRPC on IB	3840	73
Perftest	1024	88/193
XRDMA	1024	88/174
Flor	1024	84/173

Table 3: The single-core bandwidth of different network transport stacks in 100 and 200Gbps networks.

Therefore, this work provides a new choice for the community, and we expect more vendors to support UC. In addition, when migrating Flor to SmartNICs, e.g., Bluefield 2, which has multiple ARM cores and sufficient DRAM memory, thanks to Flor’s architecture that clearly separates each component, each individual component is easy to move to the BlueField’s ARM cores. The main differences from the host CPU implementation are that the ARM core is less performant than the host CPU core, and the L3 cache size of Bluefield 2 is limited, requiring the developers to optimize the system carefully. With the capability of directly operating data in the host memory introduced by Bluefield 3, the limited L3 cache size caused performance degradation can be resolved.

## 9 Related Work

**Software solutions.** We compare Flor with different network protocols. Table 3 shows the single-thread throughput between two nodes in the 100Gbps and 200Gbps network. The throughput of SNAP [33] and eRPC [20] are from the published papers. We can see that network protocols with hardware-offloaded RDMA semantics, *i.e.*, Perfest [13], XRDMA and Flor, achieve higher throughput than other network stacks. In addition, the throughput of XRDMA and Flor are comparable with Perfest, which plays the raw IB verbs without any overhead of RPC. The RDMA-based protocols can also maintain high bandwidth utilization in a 200Gbps network, where the throughput of Flor is the same with XRDMA. Though the throughput of eRPC and SNAP increases as the MTU size increases, using large MTU sizes requires a unified and standard configuration, which adds operation complexity in a complicated production environment. Besides, software solutions suffer from higher latency [33] and CPU overhead [20] without hardware

Functionalities	Lossless RDMA	Lossy RDMA	eRPC	IRMA	Flor
Transport granularity	Verbs	Verbs	MTU	Op (4KB)	Variable chunk (e.g., 4KB-64KB)
CPU Involvement	One/Two-sided	One/Two-sided	Two-sided	One-sided	One/Two-sided
Congestion control & Signal & Type	HW, ECN, rate	programmable HW, ECN+RTT, rate	SW, credit, window	SW, RTT, window	SW/HW, RTT/ECN, window/rate
Reliability & Retransmission	HW, GBN	HW, SR for RDMA Operation	SW, GBN	SW, unknown	SW/HW, intra-chunk GB0 & inter-chunk SR/GBN
PFC dependency & Loss tolerance	Yes, poor	No, high	No, poor	No, unknown	No, high
HW dependency	All RNICs	CX6-dx	DPDK/all RNICs	Customized NIC	All RNICs
Datapath zero copy	Yes	Yes	No	Yes	Yes

Table 4: Comparison on transport features of Flor and other network solutions.

acceleration.

**Hardware solutions.** To get rid of PFC, Mellanox brings up Resilient RoCE [48] and Lossy RoCE Accelerations [53] on lossless RNICs, *i.e.*, Go-Back-N-based RNICs. Resilient RoCE utilizes congestion control, *i.e.*, DCQCN, to deal with network congestion and avoid packet loss. A recent study [44] shows that the Resilient RoCE can prevent packet loss in some specific scales but still suffers unfairness from packet loss in large-degree incast events. Hardware-based lossy RDMA solutions such as Mellanox CX-5/6 [50, 52] and IRN [36] rely on strengthened hardware to run on a lossy network. They can not be deployed with CX-4 RNICs, and also lack the flexibility for users to customize each function as the implementation is highly ingrained into the hardware.

**Hardware & software co-design solutions.** RoGUE [28] designs a software congestion control for RDMA but relies on hardware reliability mechanism to recover from packet loss. It uses a large static chunk size, *i.e.*, 64KB, which needs to be revised to deal with the large-scale incast scenario. IRMA [45] is a high-performance network system that provides congestion control and reliability in software. IRMA also enables one-sided RDMA operations based on novel hardware with RDMA READ-like operation. However, it can not work on commodity RNICs, so it has little help for existing RDMA systems. Table 4 shows the clear difference between Flor and other network frameworks.

## 10 Conclusion

We present Flor, a flexible lossy RDMA framework for heterogeneous RNICs that solves a set of problems raised in production RoCEv2 clusters. These problems include PFC dependency, the interconnectivity of heterogeneous RNICs and hardware-bonded congestion control schemes. Flor onloads the reliability and congestion control function from RNICs to the software. Flor proposes a software selective retransmission for the first time at the RoCEv2 network and uses a software RTT-based congestion control to deal with the performance gap among the heterogeneous RNICs. Our evaluation of the testbed and production clusters shows that Flor achieves high performance and flexibility in many scenarios, including packet loss, heterogeneous hardware, large-scale incast, and distributed systems. Flor

also shows that the process of upgrading the existing RDMA framework to Flor has little performance impact on the running applications.

**Acknowledgments.** We are extremely grateful for our shepherd, Costin Raiciu, and the anonymous OSDI'23 reviewers for their wonderful feedback. Xiaoliang Wang is supported by NSFC No.62172204. Qiao Xiang is supported in part by the National Key R&D Program of China 2022YFB2901502, Alibaba Innovative Research Award, NSFC No.62172345, Open Research Projects of Zhejiang Lab 2022QA0AB05, MOE China Award 2021FNA02008, and NSF-Fujian-China 2022J01004. And we also appreciate for the help from Lei Yan and Shanghai Yunsilicon Technology Co., Ltd.

## References

- [1] Introducing the gaudi2 processor for training deep learning workloads. <https://habana.ai/training/gaudi2/>, 2022.
- [2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*. ACM, 2011.
- [3] InfiniBand Trade Association. Infiniband architecture specification release 1.2.1 annex a16: RoCE, 2010.
- [4] InfiniBand Trade Association. Infiniband architecture specification release 1.2.1, 2014.
- [5] InfiniBand Trade Association. Infiniband architecture specification release 1.2.1 annex a17: Rocev2, 2014.
- [6] Broadcom. Changing congestion control mode settings. <https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/adapters/Configuration-adapter/RoCE/advanced-network-configuration/changing-congestion-control-mode-settings.html>, 2022.
- [7] Broadcom. Netxtreme@-e series. <https://www.broadcom.com/products/ethernet-connectivity/network-adapters/p2100g>, 2022.

- [8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.
- [9] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.
- [10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, 2018.
- [11] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *NSDI*, 2021.
- [12] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *NSDI*, 2022.
- [13] Github. Perftest. <https://github.com/linux-rdma/perftest>, 2021.
- [14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*. ACM, 2016.
- [15] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical PFC deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2017.
- [16] Alibaba Inc. Pangu, the high performance distributed file system by Alibaba cloud. [https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud\\_594059](https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059), 2018.
- [17] Intel. Production brief for Intel® Ethernet Controller E810-CAM2/CAM1/XXVAM2. <https://cdrdrv2.intel.com/v1/dl/getContent/615503>, 2020.
- [18] Intel. Intel infrastructure processing unit (IPU). <https://www.intel.cn/content/www/cn/zh/products/network-io/smartnic.html>, 2021.
- [19] Intel. Irdma readme. [https://downloadmirror.intel.com/738730/README\\_irdma.txt](https://downloadmirror.intel.com/738730/README_irdma.txt), 2022.
- [20] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, 2019.
- [21] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.
- [22] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.
- [23] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, 2016.
- [24] Antoine Kaufmann, Tim Stampler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference*, 2019.
- [25] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *NSDI*, Renton, WA, April 2022.
- [26] Gautam Kumar, Nandita Dukkupati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. *SIGCOMM*, 2020.
- [27] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and internet-scale deployment. In *SIGCOMM*, ACM, 2017.
- [28] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: RDMA over generic unconverged Ethernet. In *SoCC*, 2018.
- [29] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

- [30] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPC: high precision congestion control. In *SIGCOMM*. ACM, 2019.
- [31] James M. Lucas and Michael S. Saccucci. Exponentially weighted moving average control schemes: Properties and enhancements. *Technometrics*, 32(1):1–12, 1990.
- [32] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-RDMA: Effective RDMA middleware in large-scale production environments. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.
- [33] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkupati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP*, 2019.
- [34] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiasheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The evolutions of the compute-to-storage networks in Alibaba cloud. In *SIGCOMM*, New York, NY, USA, 2022. Association for Computing Machinery.
- [35] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference (ATC)*, 2013.
- [36] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of ACM Special Interest Group on Data Communication*. ACM, 2018.
- [37] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.
- [38] Wiki of Linux perf command manpage. Perf wiki. [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page), 2021.
- [39] OpenCompute. In-band network telemetry in Barefoot Tofino. <https://www.opencompute.org/files/INT-In-Band-Network-Telemetry-A-PowerfulAnalytics-Framework-for-your-Data-Center-OCF-Final3.pdf>, 2019.
- [40] P4. P4. <https://p4.org/>, 2021.
- [41] Behnam Montazeri Masoud Moshref Khaled Elmeleegy Luigi Rizzo Marc de Kruijf Gautam Kumar Sylvia Ratnasamy David Culler Amin Vahdat Saksham Agarwal, Rachit Agarwal. Understanding host interconnect congestion. In *in ACM HotNets*. ACM, 2022.
- [42] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. Supercomputing on Nitro in AWS cloud. *IEEE Micro*, 2020.
- [43] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with RDMA-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016.
- [44] Alexander Shpiner, Eitan Zahavi, Omar Dahley, Aviv Barnea, Rotem Damsker, Gennady Yekelis, Michael Zus, Eitan Kuta, and Dean Baram. RoCE rocks without PFC: Detailed evaluation. KBNets, 2017.
- [45] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. IRMA: Re-envisioning remote memory access for multi-tenant datacenters. In *SIGCOMM*, 2020.
- [46] Stanford. Homasimulation. <https://github.com/PlatformLab/HomaSimulation>, 2018.
- [47] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.
- [48] NVIDIA Networking (Mellanox Technologies). Resilient roce. <https://community.mellanox.com/s/article/introduction-to-resilient-roce---faq>, 2018.
- [49] NVIDIA Networking (Mellanox Technologies). Connectx-4 lx en en card. <https://www.mellanox.com/files/doc-2020/pb-connectx-4-lx-en-card.pdf>, 2020.
- [50] NVIDIA Networking (Mellanox Technologies). Connectx-5 en en card. <https://www.mellanox.com/files/doc-2020/pb-connectx-5-en-card.pdf>, 2020.



- [51] NVIDIA Networking (Mellanox Technologies). Accelerating data center security with bluefield-2 dpu. <https://developer.nvidia.com/blog/accelerating-data-center-security-with-bluefield-2-dpu>, 2021.
- [52] NVIDIA Networking (Mellanox Technologies). ConnectX-6 Dx Ethernet SmartNIC. <https://nvdam.widen.net/s/qpszhmhpzt/networking-overal-dpu-datasheet-connectx-6-dx-smartnic-1991450>, 2021.
- [53] NVIDIA Networking (Mellanox Technologies). Mellanox lossy RoCE accelerations. <https://community.mellanox.com/s/article/How-to-Enable-Disabling-Lossy-RoCE-Accelerations>, 2021.
- [54] C. Tian, B. Li, L. Qin, J. Zheng, J. Yang, W. Wang, G. Chen, and W. Dou. P-PFC: Reducing tail latency with predictive PFC in lossless data center networks. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1447–1459, 2020.
- [55] Xingda Wei, Jiabin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015.
- [56] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memories and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2019.
- [57] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.
- [58] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. *SIGCOMM*, 2017.

## APPENDIX

### A Design Details

#### A.1 Software Reliability

##### A.1.1 Chunk sequence number

Flor has two sequence number spaces, *i.e.* global sequence number (GN in Figure 14) and reliable sequence number (RN in Figure 14). The global sequence number is a 64-bit value and all the RDMA WQEs have a unique global sequence number to identify the sequence within a QP. The reliable

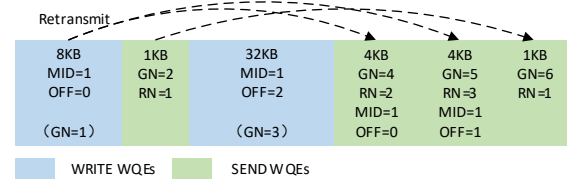


Figure 14: An example of the numbering system of Flor. An 40KB of *WRITE* WQE is split into a 8KB and a 32KB *WRITE* WQE. The 8KB *WRITE* WQE and 1KB *SEND* WQE are lost and get retransmitted. The retransmissions of 8KB *WRITE* WQE are transmitted via two 4KB *SENDS*.

sequence number is used to identify the sequence within the same type of WQEs, *i.e.*, *WRITE* WQEs and *SEND* WQEs have separate reliable sequence number space. Flor identifies the original WQEs and the retransmitted WQEs with different global numbers such that ACK information (and timestamp information carried in ACKs) is not ambiguous. The global sequence numbers for *WRITE* WQEs are maintained only at the senders and not transmitted to the receiver. The *SEND* WQEs carry both the reliable numbers and the global sequence numbers to the receiver. The retransmission WQEs share the same reliable sequence number with the original WQE. Note that Flor uses *SEND* WQE to retransmit *WRITE* WQE. This *SEND* WQE that is used for *WRITE* retransmission carries the original *WRITE* reliable sequence number, a new reliable number and a new global sequence number to the receiver such that this retransmission is able to be identified both by the sender and receiver.

The reliable sequence number of *WRITE* WQEs consists of 1-bit hint, 21-bit message id (MID in Figure 14) and 10-bit *chunk\_offset* (OFF in Figure 14). The hint bit is set when the WQE is the last WQE in the congestion window. We align the *chunk\_size* to the chunk unit size (*e.g.*, `UNIT_SIZE`). The *chunk\_offset* represents the offset of the memory address of a chunk ( $addr_c$ ) from the starting memory address of the same message ( $addr_m$ ), *i.e.*,

$$chunk\_offset = (addr_c - addr_m) / UNIT\_SIZE$$

The receiver can validate the integrity of the message by checking whether it has received data of all chunk offsets covered the message size and assemble the messages to notify the application. If using `UNIT_SIZE = 4KB`, then 10-bit chunk offset supports up to  $4KB \times 2^{10} = 4MB$  message. To support larger message in applications, users can allocate more bits for chunk offset field.

Figure 14 shows an example how this numbering system works. Here a large message of 40KB is split into 2 *WRITE* WQEs, *i.e.*, 8KB and 32KB and a *SEND* message is sent between these two *WRITE* WQEs. Each WQE has a unique global sequence number (GN) and the *WRITE* WQEs do not carry the GN to the receiver while the *SEND* does.

In the case that 8KB *WRITE* WQE and the 1KB *SEND* WQE are retransmitted. The 8KB *WRITE* WQE is split into two 4KB *SEND* WQEs, where the minimal *chunk\_size* is

4KB. Each *SEND* WQE for *WRITE* retransmission carries the original reliable sequence number of the *WRITE* WQE and has a new *SEND* reliable sequence number and a new global sequence number. Each *SEND* WQE for *SEND* retransmission carries the original *SEND* reliable sequence number and a new global sequence number to the receiver.

### A.1.2 ACK Format and Compression

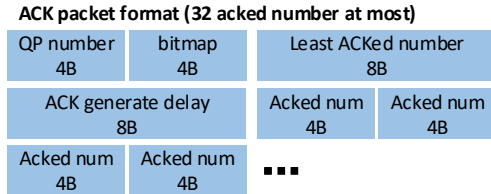


Figure 15: Software ACK format.

Figure 15 shows the software ACK format. A 32-bit bitmap (4B) in ACK packet indicates that each ACK packet signals at most 32 WQEs. A  $i^{th}$  bit set in the bitmap indicates that the  $i^{th}$  ACKed number is a global sequence number for a **SEND** WQE, otherwise, the  $i^{th}$  ACKed number is the reliable sequence number for a **WRITE** WQE. It is possible that the ACK packet contains the number of ACKed number is less than 32. As Figure 15 shows that the first ACKed number starts from 24<sup>th</sup>B and each ACKed number is 4B. The number of ACKs carried in one packet is calculated as follows:

$$(ack\_length - 24B) / 4B,$$

where *ack\_length* is the packet length of the ACK packet. For example, an ACK of packet length 40B carries  $(40B - 24B) / 4B = 4$  acked numbers. If the bitmap is 0XC0000000, then the first 2 ACKed numbers acknowledge **WRITE** WQEs and the 3<sup>th</sup> and 4<sup>th</sup> ACKed numbers acknowledge **SEND** WQEs.

We limits the acked number to be 32-bit to shorten the length of the ACK packets. The reliable sequence number of a **WRITE** WQE and the global sequence number of a **SEND** WQE will be ACKed back to the sender. Recall that the reliable sequence number is 32-bit and the global sequence number is 64-bit. Thus, we compress the 64-bit global sequence number to a 32-bit ACKed number as follows:

$$acked\_num = global\_num - least\_acked\_global\_num,$$

where the *least\_acked\_global\_num* is the smallest global sequence number in a ACK packet. The WQEs with global sequence number larger than

$$least\_acked\_global\_num + 2^{32} - 1$$

are dropped by Flor if received. This window size is large enough in practice. The silently dropped WQEs, if there are, are detected by timeout.

## A.2 RTT measurement

### A.2.1 HW/SW Clock Synchronization

The timestamps are generated by the NIC hardware clock. Except from obtaining timestamps from completions events to calculate, Flor may also need time for other usages, e.g., setting retransmission timers. However, querying current time from RNICs is a time-consuming operation (e.g., costs 1 $\mu$ s in CX-4). Thus Flor maintains a software clock based on *rdtsc*() and synchronizes the clock with hardware clock. When Flor sends or receives an operation and the clock is not corrected for 100 $\mu$ s, then Flor queries a timestamp from RNIC and update the offset when the error exceeds threshold 10 $\mu$ s. According to our observation, the successfully correct ratio (i.e., the ratio that the error exceeds 10 $\mu$ s) is less than 1%.

### A.2.2 Improve RTT Measurement Accuracy

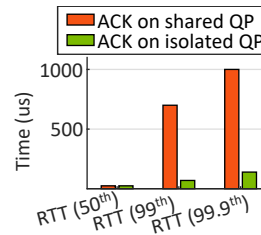


Figure 16: RTT accuracy with shared and isolated QP.

Figure 16 shows the measured RTT values with and without Flor optimization, i.e., *ACK on isolated QP* and *ACK on shared QP*, respectively. The experiment setup is the same as Figure 5 except using a larger RPC request size, i.e., 1MB.

### A.2.3 RTT Measurement for UC

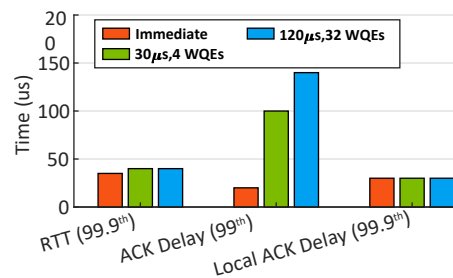


Figure 17: Evaluation of optimizations on ACK designs.

Note that Flor ACKs coalescing mechanism can also cause the ACKs being delayed. Thus, we measured the ACK delay (i.e.,  $T_3 - T_2$ ) and local delay duration (the time between  $T_4$  and ACK processing time) with different acknowledgement frequencies. Figure 17 reports the 99<sup>th</sup> percentile of these delays as the acknowledgement frequencies changes. As expected, the ACK delay increases as the number of WQEs' ACKs coalescing increases, this is because the receiver needs to wait more WQEs to finish or a timer to generate an ACK. The local delay stays the same because Flor prioritizes to poll the ACK completion queue and Flor processes one

ACK regardless the number of the number of WQEs' ACKs coalescing. Finally, the RTT measurement results show that RTT measurement accuracy does not impact by the ACK frequencies with this improvement.

# ShRing: Networking with Shared Receive Rings

Boris Pismenny<sup>†◇</sup> Adam Morrison<sup>‡</sup> Dan Tsafir<sup>†±</sup>

<sup>†</sup> *Technion – Israel  
Institute of Technology*

<sup>◇</sup> *NVIDIA*

<sup>‡</sup> *Tel Aviv  
University*

<sup>±</sup> *VMware  
Research*

## Abstract

Multicore systems parallelize to accommodate incoming Ethernet traffic, allocating one receive (Rx) ring with  $\geq 1\text{Ki}$  entries per core by default. This ring size is sufficient to absorb packet bursts of single-core workloads. But the combined size of all Rx buffers (pointed to by all Rx rings) can exceed the size of the last-level cache. We observe that, in this case, NIC and CPU memory accesses are increasingly served by main memory, which might incur nonnegligible overheads when scaling to hundreds of incoming gigabits per second.

To alleviate this problem, we propose “shRing,” which shares each Rx ring among several cores when networking memory bandwidth consumption is high. ShRing thus adds software synchronization costs, but this overhead is offset by the smaller memory footprint. We show that, consequently, shRing increases the throughput of NFV workloads by up to 1.27x, and that it reduces their latency by up to 38x. The substantial latency reduction occurs when shRing shortens the per-packet processing time to a value smaller than the packet interarrival time, thereby preventing overload conditions.

## 1 Introduction

Software systems drive Ethernet NICs through producer-consumer “rings.” A ring is a logically circular memory array shared between software and NIC, such that each ring entry points to a buffer big enough to store an Ethernet packet. Software sends data by placing packet buffers in a transmission (Tx) ring, thereby handing them to the NIC to be sent. Software receives data by removing packet buffers from a receive (Rx) ring after they have been filled by the NIC, immediately replacing them with free buffers to be used in their stead for future incoming traffic. Thus, Rx rings are always fully populated with (free or filled) buffers, whereas Tx rings are commonly partially populated or empty. Consequently, Rx rings are more memory-consuming than Tx rings.

By default, a receive ring consists of  $\geq 1\text{Ki}$  entries [11, 21, 60, 65, 86, 86, 92], each pointing to a 1500B buffer, Ethernet’s maximum transmission unit (MTU) [36]. A typical Rx ring thus requires ( $1\text{Ki} \times 1500\text{B} \approx$ ) 1.5MiB. NICs support hundreds of such rings [12, 50, 66, 71], which software uses for synchronization-free parallelism, assigning different rings to different cores in both kernel [30, 33, 68, 77, 82, 84, 90] and user [2, 8, 24, 38, 52] network stacks. The combined size of Rx buffers across all cores—henceforth denoted as  $\alpha$ —can therefore reach tens of MiBs, which might be bigger than the

last-level cache (LLC). Notably,  $\alpha$  constitutes a lower bound for the size of the NIC working set [25], as the NIC sequentially operates on all Rx buffers, one after the other, so all buffers in the circle must be used before they can be re-used. As a result,  $\alpha$  exceeding LLC capacity can be problematic for high-throughput, low-latency workloads that sustain network traffic of up to hundreds of gigabits per second (Gbps).

The problem stems from these workloads relying on data direct I/O (DDIO) [20] technology or similar. DDIO allows NIC direct memory accesses (DMAs) to read and write packets to and from the LLC while avoiding high main memory access costs [15, 29, 63, 64, 78, 79, 87, 91]. But an  $\alpha$  larger than the LLC undermines DDIO’s effectiveness, as the NIC working set is too big to be cached. Consequently, CPU memory accesses become slower, contending with DMAs for insufficient cache capacity. Accesses are thus increasingly served by main memory, making the per-packet processing time longer. This overhead translates to degraded throughput and latency of networking workloads that experience the memory as a bottleneck resource.

We exemplify this problem in §2, using run-to-completion systems [6, 26, 35, 52, 58, 69, 73] common in microsecond-scale workloads like network function virtualization (NFV). In these systems, each thread of execution consists of a loop that iteratively polls an Rx ring, receives a packet from the wire, processes it to completion (without context switches or interrupts interfering), and then sends a response.

In §3, we consider addressing the problem by reducing the size of Rx rings [91]. We find that a size smaller than 1Ki might cause a core to experience many more packet drops when the incoming traffic targets this specific core. For example, a core may sustain 2x more packets without drops using 1Ki entries instead of 128. (Increasing Rx sizes beyond 1Ki has no benefit in our workloads.) In contrast, in multicore setups, using 128 entries per Rx ring reduces  $\alpha$  without incurring additional drops, provided the incoming traffic is evenly spread between the cores, which curbs the traffic and bursts that each individual Rx ring experiences.

Motivated by this finding, in §4, we propose “shRing,” a system that alleviates the above problem by sharing a 1Ki-sized Rx ring between a set of  $N$  cores. ShRing satisfies the simultaneous needs of all sharing cores when incoming traffic is even or uneven. Sharing balances buffer usage, allowing cores that sustain heavier traffic to utilize more Rx entries at the expense of cores sustaining lighter traffic while keeping  $\alpha$  small.



ShRing is advantageous if (1) cache misses due to ineffective DDIO usage cause non-negligible overhead, and (2) the workload avoids pathologically imbalanced conditions, where a subset of the sharing cores are continuously overloaded while their peers are underloaded. (NFV studies commonly assume non-pathological conditions [4, 10, 26, 59, 63, 73, 75, 76, 97], which might indicate the system is misconfigured.) If DDIO usage *is* effective, then shRing’s synchronization overhead might degrade the performance, and if the workload *is* pathologically imbalanced, then the overloaded cores might monopolize all the entries of the shared ring. ShRing thus dynamically identifies the above two conditions, and it turns itself on or off accordingly.

When operational, shRing boosts LLC hits by shrinking the working set, which reduces the per-packet processing time ( $P_t$ ) and thus increases throughput. If shRing’s shorter  $P_t$  becomes smaller than packet interarrival time ( $I_t$ ), queuing theory dictates that ring occupancy drops from full to empty, dramatically shortening latency from linear in the ring size to essentially  $P_t$ . But even if shRing’s  $P_t$  remains greater than  $I_t$  (ring fully occupied, so latency is linear in ring size), latency still improves by a factor of  $1/N$ , as the per-core Rx ring size is effectively  $1/N$  smaller, being shared by  $N$  cores.

Shared data structures commonly underperform due to software synchronization overhead [9, 22, 26, 55, 80, 90]. ShRing reduces this overhead by avoiding synchronization when deciding which core will process which newly arriving packet. By using per-core completion rings (CRs), the NIC spreads incoming packets between cores, adding the integer index of each packet’s entry to the CR of the core that owns the packet [37]. Cores still require synchronization when notifying the NIC that ring entries can be reused. ShRing bounds this overhead by limiting  $N$ , the number of sharing cores. We use  $N=8$ , but other values may be preferable in other setups.

We explore two shRing variants. The first, “RxArr,” is a shared cyclic Rx array structured similarly to a private ring. Because it is shared, its packet buffers routinely become ready for reuse out of (array) order, as they are processed by different cores. The problem is that, for correctness, RxArr is permitted to notify the NIC that entry  $i$  can be reused only after all preceding entries (such as  $i-1$ ) are likewise made reusable. This constraint necessitates coordination between cores, which increases the overhead of synchronization.

Our second shRing variant, “RxList,” simplifies coordination by turning the shared ring into a linked list using a “next” field added to Rx entries. When storing incoming packets, the NIC follows list (rather than array) order. This change allows cores to make entries immediately available for NIC reuse; they no longer have to wait for preceding entries. We find, alas, that RxList performs poorly, as the linked list structure undermines the NIC’s ability to prefetch Rx entries, ruling this design out for the time being. We propose a modest NIC ASIC modification that resolves this problem (but prevents us from experimentally evaluating this improved design).

We demonstrate in §5 that RxArr shRing works as expected, improving NFV macrobenchmark throughput by up to 1.27x and latency by up to 38x. In §6, we experimentally show that our findings are also applicable to more traditional applications that use kernel-based TCP sockets. Finally, we discuss related work in §7 and conclude in §8.

## 2 Motivation

We begin by providing the necessary background (§2.1) and by characterizing the problem that shRing tackles, which is the increasing working set size of the NIC as compared to the LLC size (§2.2). We then experimentally demonstrate how this problem affects performance as well as shRing’s ability to address its root cause (§2.3).

### 2.1 Background

**Interacting with NICs** Software and Ethernet NICs interact via logically cyclic producer-consumer queues called *rings*. The roles of producer and consumer depend on perspective: for received (Rx) traffic, the NIC can be viewed as producing incoming packets that software consumes; alternatively, software can be viewed as producing free buffers that the NIC consumes by filling them with incoming data. Transmitted (Tx) traffic can be viewed similarly. Software chooses the ring size and allocates it in main memory. The entries of a ring are architected *descriptor* structures consisting of several fields, one of which is a pointer to packet buffer.

Software pre-allocates packet buffers for all Rx descriptors. Each buffer can hold MTU bytes ( $\approx 1500$  by default). When a packet arrives, the NIC DMA-writes it to the buffer pointed to by the *tail* descriptor of the Rx ring (“next free” index), incrementing the tail to point to the subsequent descriptor if the tail does not surpass the *head* ring descriptor. Symmetrically, software dequeues Rx packets for processing from the head descriptor (“next full” index), iteratively incrementing it so long as it does not surpass the tail; software replaces the current head’s buffer, which the NIC has just filled, “reposting” a new free buffer instead and informing the NIC about this by “ringing the doorbell” (writing to a NIC register).

Tx traffic occurs similarly, with NIC and software flipping roles (NIC responds to software actions rather than the other way around). Thus, in contrast to the Rx case, Tx ring descriptors are initially empty and therefore consume less space.

A ring’s head and tail are maintained as consumer- and producer-controlled registers, residing in NIC memory mapped I/O (MMIO) and holding ring indexes. Software may configure the NIC to trigger an interrupt when it updates a register, or it may instead poll the ring and observe changes.

The NIC distributes incoming traffic load between multiple Rx rings, and therefore between multiple cores, using receive side scaling (RSS [68], which computes a hash over the packet’s header to produce a ring identifier) or accelerated

receive flow steering (ARFS [90], which consults software-controlled packet steering tables).

**NFs** In this work, we mostly focus on improving the performance of network function (NF) workloads. NFs are packet-processing applications that were once implemented using rigid proprietary hardware middleboxes and are now increasingly implemented with software on off-the-shelf servers [4, 23, 26, 27, 53, 54, 58, 74]. Common NF examples include switches, routers, firewalls, virtual private networks (VPN), deep packet inspectors (DPI), network address translators (NAT), and load balancers (LB). Evidence suggests that nearly 60% of all data center network traffic relies on NFs [74].

To attain high throughput and low latency, NFs commonly employ a packet processing model based on kernel bypass and direct NIC access [4, 23, 27, 53, 58] as provided by, e.g., the data plane development kit (DPDK) [51]. To improve efficiency and minimize overheads, this model typically foregoes abstractions like blocking I/O, context switching, and multi-tasking. Instead, it is designed as a simple run-to-completion, polling system, which does away with costly device interrupts as means of driving networking activity. Thus, each NF thread  $T$  gets its own dedicated core and rings.  $T$  continuously polls its Rx ring, and when a packet arrives,  $T$  processes the packet, generates a response, sends the response by placing it in its Tx ring, and resumes its Rx polling.

**DDIO** High-throughput, low-latency apps like NFs benefit from Intel’s direct data I/O (DDIO) technology [20] (other processor vendors support similar technologies [5, 93]). When possible, DDIO satisfies DMA operations from the LLC rather than main memory, which is faster/cheaper and may thus improve throughput and latency. Specifically, DDIO services DMA reads from the LLC if the target data is already there, which, in addition to being faster, also reduces memory bandwidth contention. Symmetrically, DDIO can perform DMA writes directly to the LLC instead of to main memory by either overwriting existing LLC lines, if they reside in the LLC, or by allocating new lines in up to two LLC ways.

## 2.2 The Problem: I/O Working Sets

Let the *I/O working set* be the memory area that an I/O device (e.g., NIC) reads/writes via DMA in a given time interval. For NFs, this set should preferably fit in the LLC due to DDIO. An I/O-intensive workload whose I/O working set size exceeds (or even approaches) LLC capacity implies: that I/O-related data likely competes for cache capacity; that DMAs are thus increasingly served by main memory instead of the LLC; and that LLC contention and memory bandwidth bottlenecks might occur as a result [15, 29, 63, 64, 78, 79, 87, 91].

Rx ring size is a key factor in determining the I/O working set size. Recall that all Rx descriptors are pre-populated with MTU (1500B) packet buffers upon startup. Subsequently, whenever software replenishes the ring’s head descriptor with

year	Intel NIC	gen. (GbE)	max ring num. ( $r_m$ )	default size (s)	Xeon CPU	LLC	cores
2001	[40]	1	1	256	[41]	256 KiB	1
2007	[42]	10	64	512	[43]	12 MiB	4
2014	[46]	40	1536	512	[45]	38 MiB	15
2020	[49]	100	2048	2048	[48]	77 MiB	56

Table 1: The first Intel NIC model in each GbE generations shown alongside the Intel CPU launched at the same year whose LLC was the largest in that year. The number of supported NIC rings and the default ring size are increasing.

a free buffer  $B$ , the head-tail protocol (§2.1) dictates that the NIC will DMA-write a new packet to  $B$  only after the associated Rx ring tail wraps around back to  $B$ ’s position. Thus, the aggregate Rx size (denoted  $\alpha$ ) serves as a lower bound for the I/O working set. If software utilizes  $r$  Rx rings of size  $s$ , then this lower bound is  $\alpha = r \times s \times 1500B$ .

The problem that motivates our work is that  $\alpha$  grows faster than the LLC and nowadays routinely exceeds it, with Rx rings increasing both in number ( $r$ ) and size ( $s$ ). Underlying this phenomenon are, notably, the following technology trends. NIC throughput has been growing faster than CPU packet processing speed for over a decade [32, 87]. The higher bandwidth increases variability and necessitates bigger network queues [28, 47, 94]. Moreover, the ever-growing traffic volume implies that the days when a single CPU core was able to drive an Ethernet NIC to its full capacity are long gone [31]. Thus, modern systems must employ multicore parallelism [9, 22, 26, 80, 90]. NICs have therefore evolved to offer multiple Rx/Tx rings, allowing each core to interact with the NIC through its own private ring instances in isolation. We refer to this architecture as *privRing*.

To demonstrate the rapidly increasing  $\alpha$  phenomenon (along with the underlying technology trends), we collected the ring maximal number ( $r_m$ ) and default size ( $s$ ) from the datasheet and driver, respectively, of every Intel NIC model released during 2000–2022. Table 1 shows a representative summary; to conserve space, we only include the first NIC of each Ethernet generation with increasing throughput. Early 1GbE NICs supported only a single ring, but as multicore CPUs became more common, subsequent 1GbE NICs supported up to 16 rings (not shown). Later, the first generation of 10GbE, 40GbE, and 100GbE respectively introduced support for 64, 1536, and 2048 rings.<sup>1</sup> The default ring size likewise increased from 256 to 2048. Network stacks and libraries adopt similar sizes. For instance, the default Rx ring size in all sample apps in the DPDK library is currently 1024 [60].

The right side of Table 1 matches each NIC with an Intel CPU model launched at that year, whose LLC was the largest

<sup>1</sup>It makes sense for  $r$  to be much bigger than CPU core number in order to support, e.g.: per-application rings [38, 95]; per-container rings [2, 24]; a ring for every SRIOV [44] instance of every virtual CPU of every virtual machine that runs on the host machine [82, 84]; and a hypervisor ring per VM ring for fallback when flow rule offloading is not yet configured [33, 77].

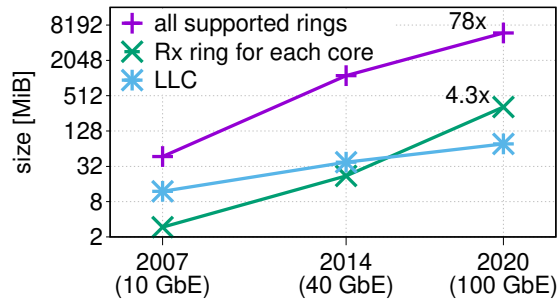


Figure 1: Aggregate Rx size ( $\alpha = r \times s \times \text{MTU}$ ) grows faster than LLC size and has already exceeded it in even the most minimalist configuration (based on data from Table 1).

at the time. Using this data, Figure 1 plots the size of the LLC and the minimal and maximal  $\alpha$  of the associated NIC. We see that the maximal aggregate Rx size (assuming all  $r_m$  supported rings are used) was always too big to fit in the LLC size in this time range. But in 2020, the aggregate Rx size of even the most minimalist configuration—just one Rx per core—became too big. This is the source of the problem.

The situation is exacerbated if considering logical, rather than physical cores (the 4.3x in the figure would have become 8.6x). We predict that this trend will continue, as upcoming NICs will bring more features (and queues), with speeds of up to 800 GbE expected in 2025 [13, 14].

## 2.3 Implications

Assume that the I/O working set size of some NF exceeds the LLC capacity and/or the LLC space it needs for satisfying DMA-writes of incoming data (constrained by DDIO to only two ways per LLC set by default) is insufficient. In this case, we claim that the overhead is significant to the point that it may be preferable to abandon dedicated private rings (privRings) in favor of shared rings (shRings), despite the synchronization cost associated with the latter.

To demonstrate, we use a synthetic FastClick NF microbenchmark configured to iteratively receive a packet, access an array, perform routing, and send the packet out [8]. The NF uses all (16) cores of our 2.1 GHz CPU, experiencing a theoretical incoming load of 200 Gbps of MTU packets (line rate), which in practice is 195.6 Gbps (due to 34B Ethernet overhead for each 1500B MTU packet). We execute this experiment using the baseline privRing, as well as three shRing variants that unify the rings of 2, 4, and 8 cores, respectively denoted as shRing/2, shRing/4, and shRing/8. (The full details of shRing are specified in §4, and the full details of the experiment are specified in §5.)

Figure 2a distills our case. It shows the average number of cycles it takes to handle one packet, breaking it down to synchronization overhead (“sync”) vs. actual processing time (“orig”). While synchronization overheads are substantial and

increase with the level of sharing, we see that it is nevertheless advantageous to pay the cost, as cycles-per-packet improves by about 4% each time we halve the I/O working set size.

The NF throughput, shown in Figure 2b, is approximately inversely proportional to cycles-per-packet (Figure 2a) as long as the CPU constitutes a bottleneck resource and line rate is not yet attained. Specifically, let  $C$  denote the average number of cycles required to process one packet, let  $hz$  ( $=2.1$  GHz) denote the cycles-per-second clock speed of the CPU, and let  $n$  ( $=16$ ) denote the number of running CPU cores, then  $n \times \frac{hz}{C}$  is the number of packets that the CPU handles per second, and so  $\text{Gbps}(C) = 1500\text{B} \times 8\text{bit} \times n \times \frac{hz}{C}$  is the throughput.

Using this equation, we can compute  $C_{bdgt}$ , the budget of per-packet cycles that the system must meet to achieve the 195.6 Gbps line rate (denoted “bdgt” in Figure 2a) as follows:  $C_{bdgt} = 1500\text{B} \times 8\text{bit} \times n \times hz / 195.6 \text{ Gbps} = 2061$  cycles per packet. Only shRing/8 meets the budget here.

We have argued that the reason underlying shRing’s improved performance is its smaller I/O working set, which curbs memory bandwidth consumption by increasing cache efficiency. This argument is directly supported by Figures 2c (memory bandwidth) and 2d (LLC misses as experienced by both CPU and NIC). In the latter figure, we see that privRing’s NIC PCIe miss rate is as high as 85%, which is why privRing’s average NIC PCIe read latency grows to  $1.45 \mu\text{s}$  (Figure 2e). Such a long PCIe latency is enough to saturate the DMA engines within the NIC (designed to hide PCIe latency with parallelism), and so it hampers the NIC’s ability to quickly process rings, which in turn generates high ring occupancy of 94% on average (Figure 2f). The implication is that, on average, each privRing packet  $P$  must wait for 966 packets ( $=94\%$  of ring size) to be processed before  $P$  is finally processed itself, which explains privRing’s high latency (Figure 2g).

In contrast, shRing/8’s occupancy is small, as it meets the  $C_{bdgt}$  budget and so its processing rate ( $\mu$ ) is larger than the arrival rate ( $\lambda$ ). Because  $\mu > \lambda$ , latency is much lower. Even when shRing does not meet the  $C_{bdgt}$  budget (the /2 and /4 variants), it improves latency, as its per-packet processing time is lower than in privRing.

## 3 Fewer or Smaller Private Rings

Conceivably, we can reduce the I/O working set size without ring sharing in two straightforward ways. One can use much smaller per-core Rx rings, or one can employ a single core (using a bigger Rx ring) as the system’s centralized “dispatcher” for all incoming traffic. Here, we briefly explain why neither is satisfactory for high-bandwidth networking applications.

The single-core, single-ring centralized dispatcher approach is used by such systems as Shinjuku [57] and Shenango [72]. It can be an effective way to reduce I/O memory consumption, and it has been shown to work well for NIC bandwidth of up to 40 Gbps. But more powerful NICs might not be served well by this approach, as the dispatcher’s



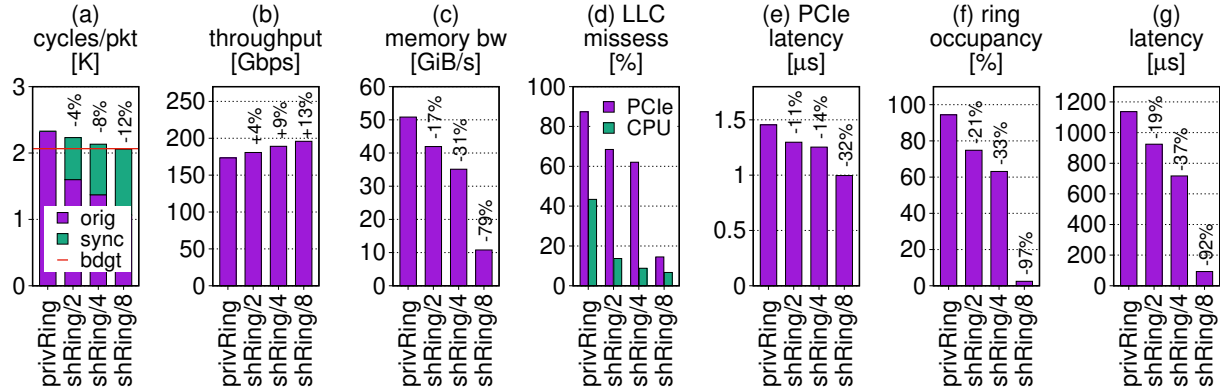


Figure 2: *ShRing*'s synchronization costs are significant but are nevertheless worthwhile, as they are cheaper than the overheads associated with *privRing*'s larger I/O working set. When *shRing*'s cycles-per-packet meet the line rate budget (a), its packet processing rate exceeds the packet arrival rate, generating low occupancy in the ring (f) and thus substantially reducing the latency (g).

limited compute capacity becomes a bottleneck [31].

The other potential approach, of reducing the size of all rings while retaining the ring-per-core design, is compatible with multicore parallelism. But we contend that the existing ring size is necessary and that reducing it has negative repercussions. To illustrate, we run the standard RFC2544 no-drop rate (NDR) test [10] with DPDK Layer-3 MTU packet forwarding (13fwd) on 8 cores. This test finds the maximum throughput attainable without loss. We run it once with traffic evenly spread across the cores (“multicore”) and again with traffic directed at one of them (“single core”).

Figure 3a shows that small rings work well for multiple cores if traffic is evenly spread between them, curbing the load and bursts that each core/ring experiences, which allows the fewer Rx buffers to cope. But small rings cease to deliver when traffic is uneven: the overloaded (“single”) core’s ring overflows and causes packet drops if it is smaller than 1Ki. In contrast, Figure 3b shows that one shared 1Ki-ring is enough to sustain optimal NDR of either 8 competing cores (each using 128 entries on average) or just one overloaded core, as *shRing* allows more loaded cores to use more Rx entries at the expense of their less loaded peers that are adequately served by fewer entries at that particular time.

## 4 ShRing’s Design and Implementation

*ShRing* is an architecture for driving high bandwidth NICs. Instead of using private per-core default-sized Rx rings, it shares each default-sized Rx ring between a set of cores. (*ShRing* leaves the Tx path unmodified.) *ShRing* can improve throughput, latency, or both, depending on the workload (§4.1).

Sharing a receive ring among cores requires us to synchronize the ring accesses of the CPU (using locks or atomic instructions), which incurs overhead compared to the synchronization-free *privRing*. *ShRing* curbs this overhead by limiting the number of cores sharing a ring to  $N$ ; we use

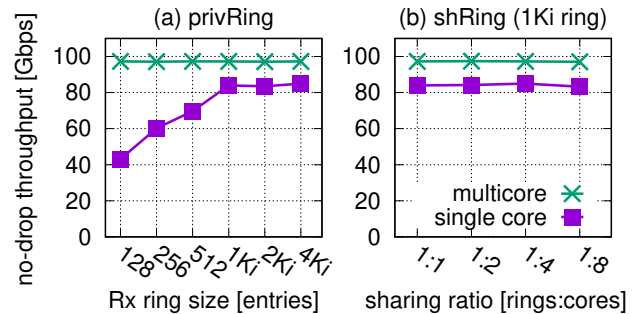


Figure 3: DPDK 13fwd no-drop rate. Small *privRings* work well when traffic is evenly spread across cores but cause drops otherwise. *ShRings* work well in both cases at a fraction of the buffer size.

$N=8$ , but other values may work better for other setups. Also, *shRing* reduces synchronization overhead by leveraging per-core completion rings (CRs) with which the NIC spreads incoming packets between cores [37], ridding them from having to compete for newly arriving packets (§4.2). As a result, *shRing*'s benefits outweigh its synchronization costs for workloads that suffer from ineffective DDIO use.

We propose two *shRing* designs that represent the ring as an array (RxArr, §4.3) or a linked list (RxList, §4.4). Both can be implemented with recent NVIDIA NICs. RxArr’s synchronization is costlier, but RxList’s interferes with the NIC’s Rx entry prefetching, so we rule it out (but propose a modest NIC ASIC modification that will fix this problem).

*ShRing* dynamically turns itself on/off depending on whether or not the workload is benefiting from it (§4.5). We describe the implementation details in §4.6.

### 4.1 Benefits and Constraints

*ShRing* can improve throughput and/or latency, depending on the workload. Next, we define the workload properties



necessary for shRing to be advantageous, and we explain the expected benefits of shRing and how it provides them. When shRing is counterproductive (necessary properties are absent), it dynamically disables itself.

ShRing is relevant only for workloads that avoid *pathological core overload*, where a subset of the sharing cores are continuously overloaded while their peers are underloaded. Pathological conditions may occur due to continuous, highly skewed per-packet processing time differences, or because of chronic incoming traffic imbalance. For reasons detailed later on (§4.5), when cores share a ring under pathological conditions, the fact that only some of them are overloaded implies that the packets of the overloaded cores increasingly and disproportionately accumulate within the ring, to the point that no room is left for packets of underloaded cores. This pathology causes new packets directed at underloaded cores to get dropped despite there being available processing capacity.

We term these conditions “pathological” because (1) they are suboptimal and may indicate the system is misconfigured, and (2) they are atypical when measuring NFV performance, as many NFV studies [4, 26, 63, 73, 75, 76, 97] and IETF benchmarking methodology [10] generate packet headers using randomization, balancing load across cores with hash-based packet spreading (e.g., RSS).

**Throughput** ShRing improves a workload’s throughput if (1) its I/O working set with privRing exceeds the LLC DDIO capacity and (2) the penalty of the resulting cache misses is non-negligible compared to the overall packet processing time. Relative to privRing, shRing multiplicatively decreases the number of rings by a factor equal to the number of cores sharing each Rx ring ( $N=8$  in our case). This decrease results in a corresponding  $1/N$  reduction of the I/O working set, possibly to below the LLC DDIO capacity. ShRing therefore mitigates and possibly eliminates the I/O-related cache miss penalty and thus enables more effective packet processing.

**Latency** ShRing improves a workload’s latency if the associated cores are saturated because packet service rate (number of packets processed per second, denoted  $\mu$ ) is smaller than packet arrival rate (number of packets arriving per second, denoted  $\lambda$ ). Latency is linear in the ring size  $s$  in this case, as queuing theory dictates that  $\mu < \lambda$  implies fully occupied Rx rings, which means every newly arriving packet waits for  $s - 1$  preceding packets to be processed. But in contrast to privRing, where each core has its own default-sized ring, shRing shares each such ring between  $N$  cores, so the “effective” ring capacity that each core experiences is  $s/N$ , which means the latency proportionally becomes  $1/N$  smaller (recall that we assume no pathological core imbalance).

Moreover, whenever shRing improves throughput, it also improves latency, as this throughput improvement stems from making the per-packet processing time ( $P_i$ ) shorter. Notably, if shRing’s shorter  $P_i$  transforms the overall service rate from slower than arrival rate (under privRing) to faster ( $\mu > \lambda$ )

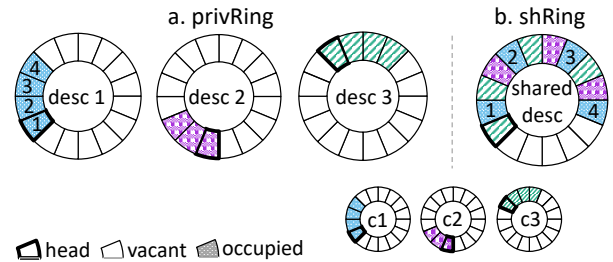


Figure 4: PrivRing (private Rx rings) vs. shRing (shared Rx ring) with  $N = 3$  completion rings.

instead of  $\mu < \lambda$ ), queuing theory says that Rx ring occupancy drops from fully to barely occupied. Namely, latency drops sharply, essentially becoming  $O(P_i)$  with shRing instead of  $O(P_i \times s)$  with privRing. This shRing property underlies Figure 2g.

## 4.2 Synchronization with Completion Rings

In principle,  $N$  cores may share a receive ring by synchronously accessing the ring’s head. But this approach creates a synchronization bottleneck [9, 22, 26, 80, 90]. ShRing sidesteps this problem by reusing RSS to spread incoming packets between different sharing *cores* (in addition to spreading them between different *rings*, which is the usual role of RSS). So when the NIC stores incoming packets in a shared ring, it communicates to each of the  $N$  sharing cores which packets belong to that core via a per-core *completion ring* (CR), as depicted in Figure 4.

A CR is a circular array in host memory. There are  $N$  CRs associated with each shared ring  $R$ : one for each core  $C$  that shares  $R$ . The CR stores indexes of  $R$ ’s packet descriptors, specifying which descriptors are ready to be processed by  $C$ . Similarly to descriptor rings, a CR has head/tail entries whose indexes reside in NIC memory. When the NIC stores in  $R$  an incoming packet  $P$  that is mapped to core  $C$ , it writes the index of  $P$ ’s descriptor to the tail of  $C$ ’s CR and advances this tail. To receive packets,  $C$  polls its CR head awaiting notification about the next available packet in  $R$ . When  $C$  removes this packet from  $R$ , it advances its CR head.

Thus, per-core CRs allow cores to poll without synchronizing with their peers. CRs negligibly increase the I/O working set size, as a CR entry occupies only a single cacheline (for storing metadata about the associated packet, such as size and header offsets). Nonetheless, CRs do not obviate the need for synchronization when a core reposts a descriptor for the NIC to consume. RxList and RxArr address this synchronization problem in different ways.

**NIC Support** Recent NVIDIA NICs already support associating multiple CRs with a shared Rx ring as part of a shared receive queue (SRQ) buffers feature [37, 61]. The motivation for this feature is reducing DRAM pinning for RDMA (see §7), as opposed to shRing’s goal of improving throughput

and latency for Ethernet.

We expect support for Ethernet Rx ring sharing among CRs to become widely available in the future, because it is included in the infrastructure datapath function (IDPF) specification [17] and the Open Compute Project NIC specification [18], which are proposed industry standards for network device interfaces.

### 4.3 Array Ring Sharing (RxArr)

In the baseline `privRing`, each core  $C$  processes and reposts descriptors of its private ring in array order, one after the other. Namely, after  $C$  processes a descriptor  $D_i$ , it reposts  $D_i$  by advancing the head of the ring past  $D_i$  to  $D_{i+1}$ , thereby indicating that  $D_i$  can be reused by the NIC to store some other incoming packet in the future.

In contrast, `RxArr shRing` implements a ring array that is shared between  $N$  cores. It therefore cannot automatically advance the ring's head in this way, as  $D_i$  might become ready for reuse before its  $k$  preceding descriptors  $\{D_j\}_{j=i-k}^{j=i-1}$ . For example, if they were assigned to cores different than  $C$  and require a longer processing time as compared to  $D_i$ . Or if RSS happened to assign all of them to some other core  $C'$ , which must now work harder than  $C$  to catch up.

`RxArr` must thus guarantee that the NIC is notified that  $D_i$  can be reused only when all preceding descriptors are also ready for reuse. For this purpose, `RxArr` maintains a bitmap with a bit per descriptor, tracking which ring descriptors between head and tail have been processed and made available for reuse. After core  $C$  consumes  $D_i$  and re-arms it with a new empty buffer,  $C$  (1) atomically sets bit  $i$  in this bitmap, (2) consults the bitmap to find the maximal contiguous sequence of descriptors available for reuse beginning at the head  $\{D_j\}_{j=head}^{j=maxContig}$ , and (3) atomically clears the corresponding bits and advances the head past them.

The drawback of `RxArr` is its synchronization overhead, as its bitmap is a shared and frequently updated data structure that requires core coordination. Also, `RxArr` is suboptimal in that it delays the reuse of descriptors made ready by some cores, if prior descriptors have not yet been processed by other cores. Conceivably, packet loss might occur under `RxArr` despite available CPU and buffer capacity. In the `privRing` baseline, in contrast, ready descriptors reside in different rings and so the NIC can reuse them as they become available.

Listing 1 shows the `RxArr` receive function, which dequeues a batch of packets for processing. It receives a shared descriptor ring (`sd_ring`), the calling core's CR (`c_ring` completion ring), and an output array of packet pointers (`pkts`) of length `len`. It returns the number of received packets. Lines 10–15 poll the CR to find the location of a ready descriptor assigned to the calling core and store the descriptor's buffer in the output array, replacing this buffer with a new one. Lines 16–22 mark received descriptors in the shared bitmap (`sdr->bitmap`) while batching updates within 64-bit

```
1 #define BIT(x) (1 << ((x) & 63))
2 #define WORD(x) ((x) >> 6)
3 #define ISSET(bmp, x) \
4     (bmp[WORD(x) & (bmp->size - 1)]) & BIT(x))
5 int shRing(sd_ring *sdr, c_ring *cr,
6           void **pkts, int len) {
7     int rcvd = 0, lidx = -1;
8     uint_64t lbits = 0
9     while (rcvd < len) {
10        c_ring_ent *cre = get_cre(cr);
11        if (cre == NULL)
12            break;
13        int idx = cre->idx;
14        pkts[rcvd++] = sdr->desc[idx].buf;
15        sdr->desc[idx].buf = alloc_buf();
16        if (lidx == -1) lidx = WORD(idx);
17        else if (lidx == WORD(idx)) {
18            atomic_or(&sdr->bitmap[lidx], lbits);
19            lidx = WORD(idx);
20            lbits = 0;
21        }
22        lbits |= BIT(idx);
23    }
24    if (rcvd == 0) return 0;
25    if (lbits != 0)
26        atomic_or(&sdr->bitmap[lidx], lbits);
27    cr->ci += rcvd;
28    *cr->doorbell = cq->ci;
29    lock(sdr->lock);
30    while (ISSET(sdr->bitmap, sdr->ci) != 0) {
31        setb = ffs(~sdr->bitmap[WORD(sdr->ci)]);
32        atomic_clear(&sdr->bitmap[WORD(sdr->ci)],
33                    setb - 1);
34        sdr->ci += setb - 1;
35    }
36    *sdr->doorbell = sdr->ci;
37    unlock(sdr->lock);
38    return rcvd;
39 }
```

Listing 1: `RxArr` shared ring receive code.

words. This is done using atomic instructions, as other cores may be concurrently setting/clearing other bits in the bitmap. Line 24 handles the corner case of an empty CR. Lines 25–26 handle the remaining accumulated bitmap updates after exiting the loop. Lines 27–28 ring the CR's doorbell.

Lines 29–37 identify the maximal contiguous sequence of descriptors beginning at the ring head that is available for reuse, notifying the NIC about them. These operations are performed under a lock to guarantee the atomicity of (1) inspecting and modifying the bitmap and of (2) notifying the NIC. Line 31 uses the find-first-set instruction to identify the contiguous set bits. Lines 32–33 atomically clear them. Finally, Line 34 advances the ring's head (consumer index, `sdr->ci`) accordingly, and Line 36 writes the updated head to the shared ring's doorbell.

### 4.4 Linked List Ring Sharing (RxList)

`RxList` is a `shRing` design that alleviates `RxArr`'s bitmap coordination problem, eliminating the requirement to repost

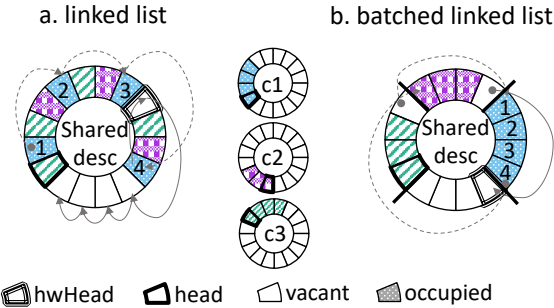


Figure 5: RxList vs. Batched RxList designs for one shared ring with three completion rings. In (b), the batch size is 4.

descriptors in array order. To this end, RxList represents the empty packet buffer descriptor queue as a linked list. The NIC correspondingly follows list order when storing incoming packets. The list itself is overlaid on the Rx descriptor array, with each descriptor holding a “next” field pointing to the next list item. (Linked list functionality is part of the SRQ feature [7].) Initially, each descriptor points to the subsequent descriptor in the array. But as packet processing occurs and cores process and repost descriptors out of array order, the descriptor order in the list changes. We denote the first and last descriptors in the empty descriptor list as *hwHead* and *hwTail*, respectively, to distinguish them from the “head” and “tail” used in the rest of the paper to describe the first and last descriptors holding packets.

Figure 5a depicts RxList’s structure using three cores sharing a single Rx ring. Observe that RxList’s descriptor ring entries are not contiguous: there are multiple non-vacant descriptors in the array between *hwHead* and its successor vacant descriptor in the list, which is impossible in an array-based design. The figure also shows dashed links between non-vacant descriptors. These represent the order in which these descriptors were filled by the NIC, i.e., their order in the list when they were vacant.

We now detail RxList’s receive flow, whose code is shown in Listing 2. The function’s inputs and outputs are the same as RxArr’s receive function. Lines 5–10 batch packets for processing exactly as in RxArr: the completion ring is polled to find the location of ready descriptors, each such descriptor’s buffer is stored in the packet output array, and the descriptor’s buffer is replaced with a new buffer. Lines 11–13 are unique to RxList: they link dequeued descriptors one after the other, creating a linked list that will eventually be appended to the tail of the empty descriptor list. Lines 15–17 are again standard functionality. First, the case of an empty completion ring is checked, and then the core’s completion ring head (denoted *ci*, or consumer index) is updated, including a notification to the NIC via a doorbell MMIO write. Lines 18–24 are again new to RxList. They lock the shared descriptor ring to atomically (1) append the new list created in lines 11–13 after the tail of the list and (2) notify the NIC, via a doorbell

```

1 int ll_rcv(sd_ring *sdr, c_ring *cr,
2         void **pkts, int len) {
3     int idx, rcvd = 0, myhead, *iptr = NULL;
4     while (rcvd < len) {
5         c_ring_ent *cre = get_cre(cr);
6         if (cre == NULL)
7             break;
8         idx = cre->idx;
9         pkts[rcvd++] = sdr->desc[idx].buf;
10        sdr->desc[idx].buf = alloc_buf();
11        if (iptr == NULL) myhead = idx;
12        else iptr->next = idx;
13        iptr = &sdr->desc[idx];
14    }
15    if (rcvd == 0) return 0;
16    cr->ci += rcvd;
17    *cr->doorbell = cq->ci;
18    lock(sdr->lock);
19    int prevtail = sdr->hwTail;
20    sdr->desc[prevtail].next = myhead;
21    sdr->hwTail = idx;
22    sdr->ci += rcvd;
23    *sdr->doorbell = sdr->ci;
24    unlock(sdr->lock);
25    return rcvd;
26 }

```

Listing 2: RxList (linked list) shared ring receive code.

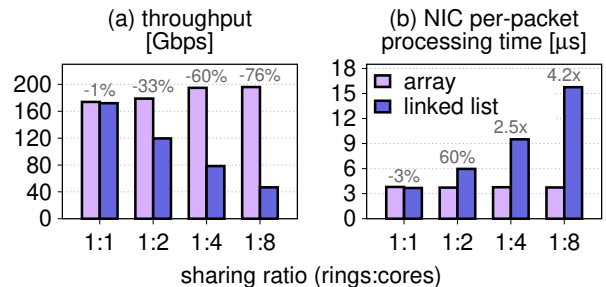


Figure 6: Although conceptually more suitable for sharing, RxList interferes with descriptors’ contiguity, hampering their prefetching and thus degrading performance. (Labels show List to Arr ratio.)

write, of the number of descriptors with empty buffers that are appended to the list. Finally, line 25 returns the number of received packets.

**Prefetching Problem** We find that RxList neutralizes descriptor prefetching, an important NIC performance optimization. Because descriptor rings are typically stored contiguously, the NIC reads sequences of contiguous descriptors in a single PCIe read transaction and caches valid descriptors in NIC memory to improve throughput and reduce latency for subsequent packets. When descriptors are linked out of array order, the NIC fails to find the next descriptor on the list in its on-NIC cache, resulting in more descriptor DMA reads being required.

Effective descriptor prefetching is critical for high PCIe-based NIC performance [70], and even more crucial for shRing. In *privRing*, a descriptor cache miss on some ring

does not stall incoming traffic destined to other rings, but with shRing there are fewer rings and so more traffic is stalled.

To demonstrate this effect, we evaluate the performance of various descriptor ring to core sharing ratios. We compare RxList to RxArr, in which the NIC follows descriptor array order when storing packets. We run the synthetic NF (from §2.3) on all cores and try to process traffic at line rate.

Figure 6a shows the throughput achieved by both designs. When there is no sharing, then RxList, RxArr, and privRing (not shown) perform similarly ( $\approx 2\%$ ). This is expected since in this case, all approaches maintain ordering within the single descriptor ring. However, as we decrease the ring to core ratio, linked list descriptors become reordered and RxList’s throughput declines sharply as sharing increases: 33% for 1:2 sharing ratio and 76% for 1:8 sharing ratio.

Figure 6b shows how costly out-of-order descriptors are, motivating RxArr. Specifically, we report the NIC’s internal packet processing time, and see that for linked lists this time grows as more cores share a descriptor ring: from 3.7  $\mu$ s at 1 core per ring to 16.3  $\mu$ s at 8. In contrast, RxArr performance remains the same regardless of the sharing ratio.

**Prefetching Solution** We propose *batched RxList*, a shRing design that obtains RxList’s resiliency against pathological core overload conditions without damaging the NIC’s performance. Batched RxList amortizes the cost of locking and descriptor reordering in RxList by batching packets to descriptors. In this design, depicted in Figure 5b, each RxList descriptor points to a buffer that can hold multiple packets. For each RxList, the NIC stores new packets destined to a core via the same descriptor used to store previous packets for that core, provided that room remains in the descriptor’s packet buffer. Only once this descriptor “fills up” will the NIC consume a new descriptor from the list and start storing incoming packets for that core in the new descriptor’s buffer. To perform this batching, the NIC caches the last Rx descriptor used for each CR associated with the RxList. The NIC thus effectively maintains per-core “mini hwHeads” pointing to each core’s current descriptor.

The benefit of the batched RxList design is twofold. From the NIC’s perspective, batching packets in descriptors and caching the descriptors reduces the importance of descriptor prefetching, as packets destined to a core experience a single cache miss per batch. From the cores’ perspective, batching reduces RxList synchronization, as locking the RxList to re-post a descriptor is now guaranteed to occur only once per batch, instead of potentially once per packet.

Although recent NICs support batching multiple packets in a single large descriptor buffer [3], batched RxList requires NIC ASIC modifications to support a list consisting of such descriptors. Therefore, we cannot evaluate batched RxList. We present this design to underscore that RxList’s tradeoffs are likely not fundamental and are caused by current NIC ASIC limitations, which can be fixed.

## 4.5 Dynamic ShRing

We propose a dynamic approach that switches between privRing and shRing during run time, depending on which architecture is more beneficial at the moment. Our goal is to disable shRing if the workload experiences pathological core overload or if it is not bottlenecked on I/O-related cache misses. We describe the heuristic we currently use to identify these conditions. We leave improving the precision and robustness of the heuristic for production use to future work.

**Pathological Overload** Pathological overloaded conditions can make overloaded cores monopolize ring descriptors. If continuous, high per-packet processing time differences are such that the packet service rate of overloaded cores is smaller than their packet arrival rate, queuing theory dictates that the Rx ring eventually becomes fully occupied with their packets. If incoming traffic is chronically imbalanced, large batches of packets destined to overloaded cores can arrive and occupy most if not all the descriptors.

In both of the above scenarios, overloaded cores invoke their ring’s receive function less frequently than underloaded cores. This is clearly the case for cores overloaded due to high per-packet processing time, but also happens if overload is due to incoming traffic imbalance. In this case, an overloaded core’s receive call produces a large batch of packets, which takes the core longer to process before returning to the ring to dequeue more packets. We detect overloaded cores based on this behavior, as explained below.

**I/O-Related Cache Miss Significance** Recall that under non-pathological conditions, a workload will benefit from shRing if (1) its I/O working set with privRing exceeds the LLC DDIO capacity and (2) the penalty of the resulting cache misses is non-negligible (§4.1). We associate (1) with high memory bandwidth utilization and (2) with high networking throughput.

**Heuristic** We measure throughput, memory bandwidth, and time between subsequent calls to the receive function and record the results in a sliding window of 16 entries. When more than half of throughput and memory bandwidth measurements exceed a predefined threshold while no core is overloaded (calls receive infrequently compared to other cores), we switch from privRing rings to shRing rings. To switch back from shRing to privRing, we wait until  $\frac{7}{8}$  of measurements are below the threshold

To switch between privRing and shRing, we pre-program two sets of RSS tables, which are NIC data structures used to steer incoming packets to descriptor and completion rings based on packet headers. Each RSS table set points to its own set of rings, i.e., privRing and shRing. Then, based on the heuristic’s decision, we update NIC steering rules to redirect packets to the appropriate RSS table set. After switching, before we begin polling the new rings for packets, we drain remaining packets from the previous ring set.



## 4.6 Implementation

Our implementation of RxArr and RxList targets 100 GbE NVIDIA NICs with unmodified ASICs. We initially relied on firmware patches to expose ring sharing mechanisms, originally aimed for InfiniBand RDMA (see §7), for Ethernet use. However, NVIDIA NIC firmware now makes these mechanisms generally available.

We implement our designs with 2039 lines of code (LOC) in the NVIDIA DPDK driver and only 137 LOC in DPDK’s core. We leverage DPDK’s command line driver options to enable the desired ring sharing mechanism and to specify how many cores share each ring. This approach enables unmodified DPDK-based applications to benefit from shRing.

Dynamic shRing is implemented in a dedicated thread that runs every 10 ms on a separate core which polls Intel PCM [39] counters for PCIe generated memory bandwidth and NIC byte and packet counters. We expose PCM counters through a library that we link with DPDK; the library is 116 LOC and the code using it in DPDK is 330 LOC. As the threshold for switching from privRing to shRing, we use throughput greater than 170 Gbps, memory bandwidth greater than 25 GiB/s, and the standard deviation between calls to Rx functions being at most 32x larger than the median (where 32 is the maximum packet batch that shRing’s Rx functions can return). We experimentally find that these values provide good results for the NFs we tested.

## 5 Evaluation

We evaluate shRing’s effectiveness using synthetic microbenchmarks as well as NAT and LB macrobenchmarks. We measure the gains obtained with shRing’s efficient I/O working set utilization in both non-pathological and pathological conditions (§4.1) under 200 GbE load.

### 5.1 Methodology

**Experimental Setup** Our setup consists of two Dell PowerEdge R640 servers, connected back-to-back via two pairs of 100 GbE NVIDIA ConnectX-5 NICs with pause frames disabled. One server is the evaluated system and the other is the load generator. Both servers have 16-core 2.1 GHz Xeon Silver 4216 CPUs, 128 GiB (=4x16 GiB) 2933 MHz DDR4 memory, and a 22 MiB LLC that consists of 11 ways. They run Ubuntu 18.04 (Linux 5.4.0) with hyperthreading and Turbo Boost disabled. The kernel is configured to isolate CPUs from the OS scheduler, use 1 GiB hugepages, disable power saving states, and disable microarchitectural side channel mitigations.

On the load generator machine, we run the stateless Cisco T-Rex packet generator [16], which we modify to improve latency measurement accuracy from 10–100 $\mu$ s to 1 $\mu$ s [81]. Unless specified otherwise, we use default application settings: 1024 descriptor Rx and Tx rings and 2 DDIO LLC

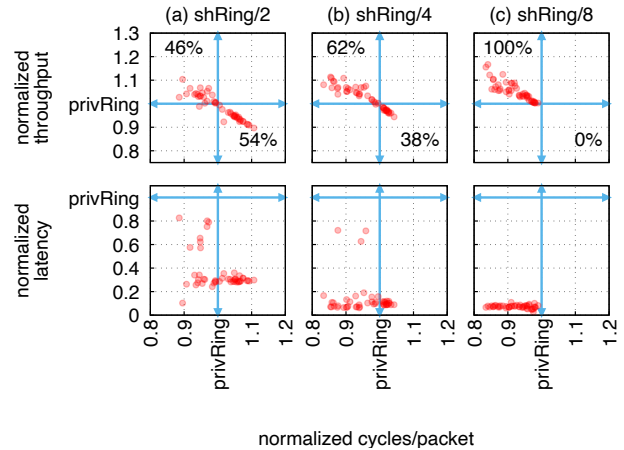


Figure 7: Normalized performance of shRing to privRing for NFs with varying memory intensity: shRing/8 improves performance in all cases. (Labels show percentage of NFs in quadrant.)

ways, and we run application logic on all 16 of the available CPU cores—8 cores per NIC. All the results presented are trimmed means of ten runs; the minimum and maximum are discarded. The standard deviation is always below 5%.

**Measurement Tools** We measure cycles per packet by modifying applications to record cycle counters, cache hit rate using Linux perf, Tx ring occupancy by comparing completion ring producer and consumer indexes, PCIe latency using NVIDIA Mellanox Neo-host [67], and memory bandwidth and PCIe hit rate using Intel PCM [39].

**Ring Mechanisms** We compare between privRing; non-dynamic array ring sharing (RxArr) between 8 cores—the maximum possible on a CPU with 16 cores and 2 NICs—which we denote “shRing/8;” and a small privRing configuration whose aggregate descriptor count equals that of shRing/8, i.e., 128 entries per ring when shRing/8 uses 1024 entries per RxArr. We remark that small privRing is impractical since it imposes loss when traffic is bursty, as shown in §3. We show it for a thorough comparison between privRing and shRing.

### 5.2 Non-Pathological Conditions

We show the benefits of using shRing under high load without pathological core overload conditions. Specifically, we evaluate (1) synthetic NFs with varying memory intensity and cache pressure; (2) NAT and LB performance; and (3) MICA key-value store performance.

For NFs, we use large 1500B UDP packets sent at 200 Gbps to stress the I/O working set, and select packet 5-tuples at random to spread the load across cores.

**Memory Intensity** To explore shRing performance with NFs of various memory intensity, we run FastClick’s synthetic WorkPackage module [8] which receives a packet, performs routing, followed by a number of random memory reads from

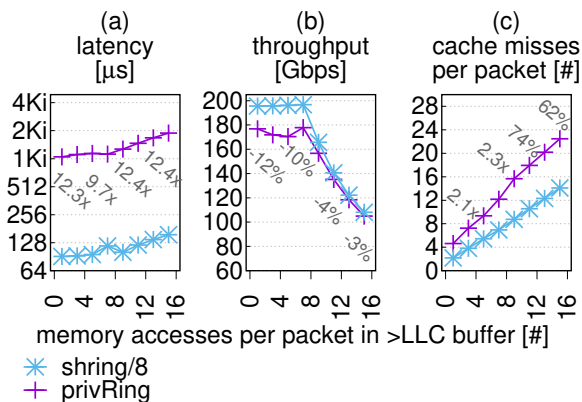


Figure 8: High cache pressure decreases performance for shRing and privRing. (Labels show privRing to shRing ratio.)

a buffer, and then sends the packet out. We modify WorkPackage to optionally read or overwrite packet payload.

We test 60 configurations: randomly reading 1, 2, 4, 8, or 12 times from a 1MiB, 10MiB, 20MiB, or 40MiB buffer (corresponding to L1, L2, LLC, and larger than LLC sizes), while packet payload is either untouched, read, or overwritten.

For each configuration, we plot shRing throughput, latency, and cycles per packet normalized to privRing; Figure 7 shows the results. We find that throughput and latency improve with descriptor sharing ratio: shRing/8 obtains the best throughput and latency followed by shRing/4 and then shRing/2. Moreover, shRing/8 always outperforms privRing (all are above the horizontal line), while shRing/4 and shRing/2 underperform privRing for 54% and 38% of the most memory intensive configurations, respectively. Exploring the configurations where shRing/2 and shRing/4 are less successful than privRing, we find that they consist of 3/16 and 11/16 NFs that read packet payload, and 5/16 and 6/16 configurations that overwrite payload, for shRing/2 and shRing/4, respectively.

**Workload Cache Footprint** We explore shRing effectiveness as the workload’s cache footprint grows. We use the aforementioned synthetic NF with 1–16 random memory accesses per packet in a 40 MiB array. Figure 8 shows the results. ShRing mitigates I/O working set induced cache misses, improving application cache hit rates by up to 2.1x, which translates to up to 13% higher throughput and up to 13.1x lower latency. As the workload’s cache footprint grows, so does CPU processing time per packet, so eventually cores exceed the CPU cycle budget needed for line rate processing. Both throughput and latency degrade as a result. As the number of processed packets thus decreases, the I/O working set induced cache stress decreases too, and so the gap between cache misses per packet in privRing and shRing shrinks.

**NAT and LB** We use two stateful FastClick NFs as macrobenchmarks: NAT and LB, which cache up to 10M flows using per-core cuckoo hash tables. NAT consistently remaps

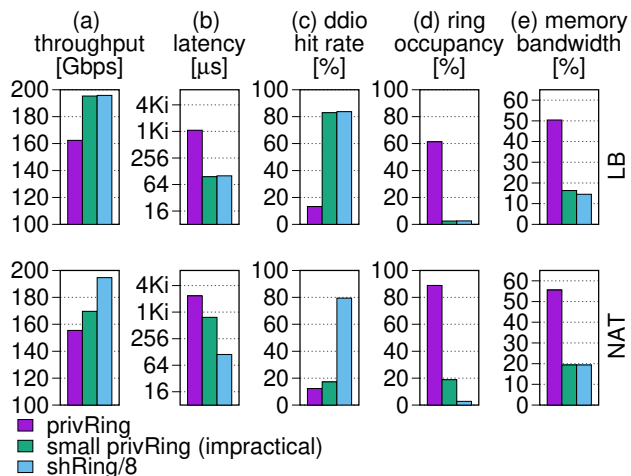


Figure 9: LB and NAT performance at 200Gbps load.

and rewrites incoming and outgoing packet IP packet headers. LB matches each flow with one of 32 destination servers, maintaining the match for each flow and making new matches with a round-robin policy. NAT is more memory intensive than LB, as it uses two cache entries per flow (one for each direction) while LB uses only one

We show results with a load of 200 Gbps. Results with speeds greater than 170 Gbps are similar, while lower speeds show no difference in throughput and less than 5 μs in latency in favor of privRing due to the synchronization overhead of shRing. The results we show are for the default Rx ring size (i.e., 1024), results for other ring sizes are similar in nature.

Figure 9 depicts the resulting (a) throughput, (b) latency, (c) ring occupancy, (d) PCIe (DDIO) miss rate, and (e) memory bandwidth. The results show that shRing/8 outperforms privRing in throughput and latency, which is consistent with previously presented microbenchmarks. This happens because at high offered load the I/O working set starts contending with the CPU for LLC space and memory bandwidth, which slows CPU packet processing. CPU slowdown, in turn, causes ring occupancy to grow, which increases latency (as explained in §2.3).

We expect small privRing to perform similarly to shRing/8, and indeed this is the case for LB, but surprisingly small privRing NAT performance is worse than shRing. For NAT, small privRing has a notably lower DDIO hit rate and higher ring occupancy. We speculate that the root cause is that shRing reposts buffers slower as it waits for other cores to make progress, and therefore its working set is slightly smaller because less buffers are exposed to I/O.

ShRing achieves high performance because it shrinks the I/O working set size to fit in the default DDIO portion of the LLC (i.e., two LLC cache ways). When disabling DDIO, namely forbidding NIC DMA writes from allocating ways within the LLC, all ring types achieve only 150 Gbps through-

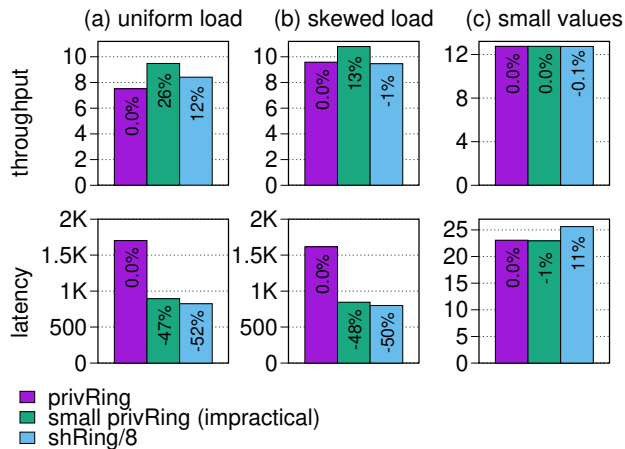


Figure 10: ShRing benefits the MICA key-value store with large I/O working sets, non-pathological load imbalance, and high load.

put and 1.3  $\mu$ s latency, which is 3% and 27% lower than privRing and shRing/8 with default DDIO (not shown in the figure). When assigning all LLC ways to DDIO, privRing performance matches shRing for LB, but it is insufficient for the more memory intensive NAT application, which uses twice as much state and whose throughput improves by less than 5% (also not shown).

**Key-Value Store** We use the MICA key-value store [62] to show that shRing is applicable beyond NFs and to highlight how workload conditions impact shRing’s effectiveness. We run MICA on 8 cores using a single 100GbE NIC, with 128 B keys and 1KiB values.

Figure 10a shows the results of a workload with 95% set operations, uniformly distributed among all cores, at the highest possible request rate. This workload satisfies the conditions that make shRing beneficial (§4.1)—i.e., (1) no pathological core overload, (2) a large I/O working set, and (3) non-negligible penalty of I/O-related cache misses. ShRing improves MICA throughput by 12% and reduces latency by 52% in this workload; small privRing shows the potential throughput gain from reducing the I/O working set, without shRing’s synchronization cost.

Figure 10b changes the workload’s traffic spread, making it imbalanced (Zipf distribution of skewness 0.99). Consequently, shRing reduces throughput by 1% over privRing but still improves latency by 50%. Figure 10c shows the initial workload but with 128B values, which makes the I/O working set small. ShRing makes no throughput improvement and increases latency by 11%. We obtain similar results when lowering the request rate of Figure 10a’s workload (not shown). In both these cases, shRing adds synchronization overhead which is not offset by I/O working set related improvements, either because the I/O working set was small to begin with (Figure 10c) or because the penalty of I/O-related cache misses is negligible (low load).

### 5.3 Pathological Conditions

This section demonstrates shRing’s sensitivity to pathological core overload, where one of the shared ring’s cores is continuously overloaded compared to the rest. We evaluate shRing/8, referred to as “shRing” here, as well as dynamic shRing/8 (denoted “dshRing”) and its ability to gracefully fall back to privRing in pathological conditions. We evaluate two causes for pathological conditions: variability in processing and variability in incoming packet distribution among cores. We also evaluate NAT and LB throughput when offered load switches from non-pathological to pathological over time.

**Processing Variability** In this experiment, we choose a target core per NIC and control its processing speed by varying the number of memory accesses it performs per packet while all other cores run the synthetic workload described in §2.3.

Figure 11a depicts the resulting throughput. When the target core’s packet processing is fast, shRing and dshRing throughput is 12% higher than privRing, but as the core’s processing slows down, shRing throughput declines to 58% lower than privRing. In contrast, dshRing notices that one core is slowing down shRing and switches to privRing, thereby avoiding performance degradation.

Figure 11b explains the observed throughput, by showing the time shRing Rx descriptors wait for co-sharing core bitmap updates before being handed back to the NIC. We present only shRing and dshRing, because privRing does not have such delays. In shRing, slow processing on the target core can delay co-sharing cores from making their processed Rx descriptors available for NIC reuse. This effect is negligible when the target core makes less than 100 memory accesses per packet, but subsequently, descriptor wait time increases dramatically (up to 257  $\mu$ s) and throughput decreases.

**Traffic Variability** Here, we choose a target core per NIC and vary the percentage of packets directed to it up to 30%. All cores run the synthetic workload. We direct 64 B packets at the target core and 1500 B packets at the others, so that even when receiving 30% of the packets, the target core’s incoming traffic is < 3% of total incoming throughput. This means that in principle, the target core’s behavior should have negligible effect on overall throughput.

Figure 12a shows the throughput in practice. When the packet load on the target core is less than 15%, shRing outperforms privRing and dshRing’s heuristic correctly enables shRing. But as load exceeds 15%, the targeted core becomes overloaded and so shRing throughput declines by up to 54%. In contrast, privRing throughput declines by only 3%, since other cores are not affected. DshRing’s heuristic identifies when the achieved throughput is too low and that it will not be improved by shRing, and thus switches to privRing.

Figure 12b shows that as with processing variability, shRing’s throughput decreases because the unloaded cores’ Rx descriptor reposting is delayed by the overloaded core.



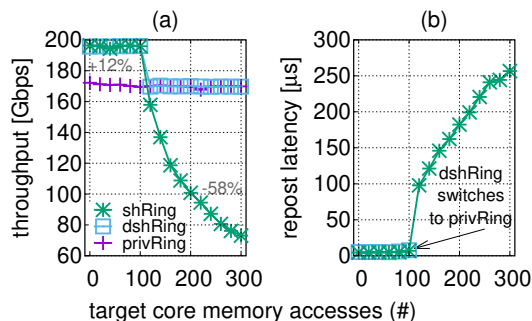


Figure 11: When incoming packet rate is fixed, processing variability in one core (e.g., due to increased number of memory accesses) might degrade shRing’s throughput and delay descriptor reposting in peer cores. Dynamic shRing falls back on privRing when this happens.

Figure 12c presents the ratio of packets successfully processed by the target core out of all packets. While shRing maintains the target core’s ratio of outgoing to incoming packets, the cost is that as more packets target this core, shRing delays receiving on other cores. This results in drops of the 1500 B packets when the target core is overloaded, and thus throughput declines. In contrast, privRing drops excess packets that exceed the target core’s processing capacity, and as a result it has at most 17% outgoing packets on the target core.

**Handling Variability with Dynamic ShRing** We run an experiment where the incoming load switches from non-pathological to pathological after 20 seconds. Figure 13 shows NAT and LB throughput sampled every second. DshRing initially uses privRing, but as load increases, it identifies high throughput and memory bandwidth with no overloaded cores and switches to shRing. At 20 seconds, we reconfigure the load generator to send a pathological load, which overloads cores and decreases throughput. DshRing identifies the drop in throughput and switches back to privRing. Consequently, dshRing achieves good performance in both.

## 6 Kernel-Based TCP Sockets

Our implementation and evaluation focus on NFV workloads, which typically bypass the operating system (OS) networking stack and the socket abstraction. This section explores the potential benefit to socket-based TCP applications from deploying shRing in the Linux networking stack.

Concerns about the effectiveness of a shRing-based NIC OS driver are that (1) application working sets may be too large for shRing’s improved DDIO utilization to matter and (2) even if not, small private rings might not lead to packet loss in the Linux kernel, as opposed to with DPDK.

Because our shRing prototype is DPDK-based, we cannot directly evaluate shRing in the Linux kernel. We therefore use “small privRing” as a proxy, to show the benefit of

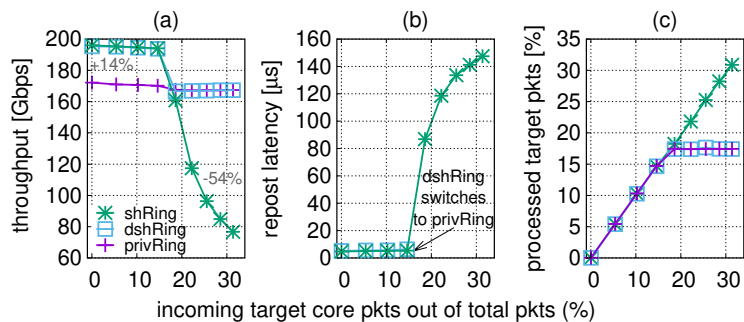


Figure 12: When variability manifests as increased rate of packets targeting one specific core (x axis), at some point, it prolongs the latency of peer core descriptor reposting (b); at this point, performance degrades (a) as the target core processing can no longer match the volume of incoming traffic (c).

reducing the I/O working set in the Linux kernel. We run Netperf [56] microbenchmarks to show that: (1) smaller I/O working sets can improve performance of a socket-based application and (2) 1Ki-sized rings are necessary to handle burstiness in the kernel.

**Pros of Smaller I/O Working Sets** We measure Netperf TCP request-response throughput (sum of Rx and Tx). We use 16 cores and two NICs with two threads per core (one per NIC). For symmetry, we use the same ring size on both sides. In all experiments, the CPU is not the bottleneck.

Figure 14a shows the throughput obtained for 64KiB requests and various response sizes. In this setting, small rings outperform large rings by up to 10%. But when the size of the request and the response are equal (Figure 14b), the results become less conclusive, e.g., for 1KiB messages throughput is almost the same for both ring sizes, and for 4KiB messages, the small ring’s throughput is 5% less than the default.

**Cons of Small Private Rings** We measure Netperf TCP stream throughput for various private ring sizes, with traffic either directed at a single core or evenly spread among 8 cores. Figure 15 (similarly to Figure 3) demonstrates that small rings work well for multicore TCP traffic, as the spread of load curbs the bursts each individual core/ring experiences. However, a single ring smaller than 1Ki overflows and causes drops, which cause TCP to back off and thus degrade throughput.

## 7 Related Work

**Efficient LLC Utilization** DDIO enabled platforms allow NICs to access data faster via the relatively small LLC. Many previous works, unrelated to ring sharing, proposed techniques to improve DDIO efficiency: (1) using small private rings to reduce the I/O working set [91]; (2) placing packets in LLC slices closest to the target processing CPU core [29]; (3) eliminating interference between applications and I/O devices when partitioning the LLC [96]; (4) placing only packet



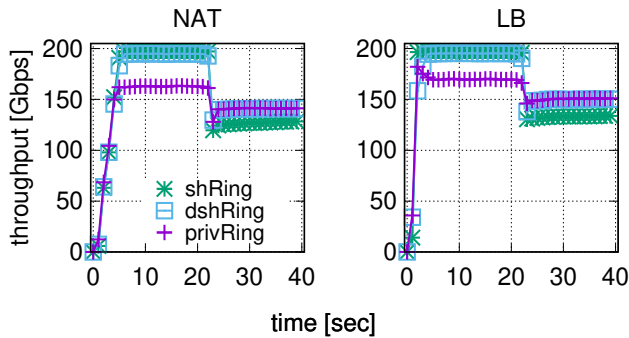


Figure 13: NAT and LB throughput when switching from a non-pathological to a pathological workload.

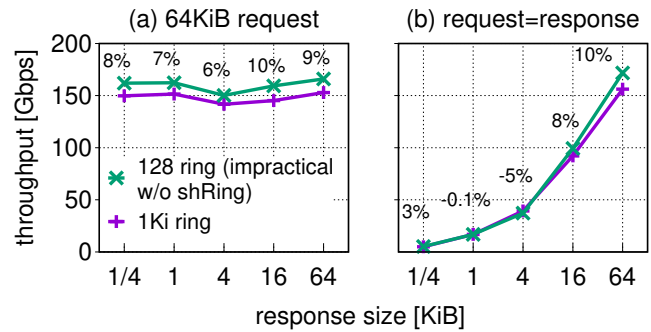


Figure 14: Netperf TCP\_RR throughput with (a) 64KiB requests for various response sizes and (b) equal request and response sizes. Small rings work better as they reduce the I/O working set.

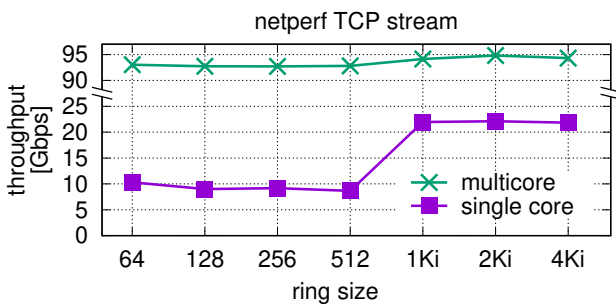


Figure 15: Netperf TCP stream throughput. Small rings work well when traffic is spread across multiple cores but cause drops otherwise.

headers in the LLC to reduce LLC contention [34, 79, 83]; and (5) modifying CPUs to prefetch DDIO-written data into mid-level caches and to invalidate data without writeback when possible to conserve memory bandwidth [1]. We show that small private rings are insufficient and propose a ring sharing mechanism that is symbiotic with the last four techniques.

**Sharing Within a Core in Software** Linux `io_uring` “automatic buffer selection” [19] lets applications pre-register buffers and later consume these via `read/recv` system calls for different file descriptors. Similarly, buffers posted to shRing are pre-registered and later assigned to cores at packet arrival time. But unlike `io_uring`, shRing operates between software and hardware.

**Sharing Within a Core in Ethernet NICs** When a single core and privilege level have multiple NIC rings, sharing their buffers and CRs to conserve resources is desirable. For example, SRIOV NICs expose a ring per VM on the hypervisor to receive packets missing hardware virtual switching rules, allowing the hypervisor to install matching rules [33, 77]. As the number of VMs exceeds the number of cores, multiple such rings must share a core. To optimize this, NVIDIA NICs

recently started sharing ring buffers and CRs within each core [61] via the same firmware changes that we used, which are now publicly available. ShRing, in contrast, shares rings between cores.

**Sharing Between Cores in RDMA** RDMA applications typically employ queue pairs (QPs) with dedicated buffers to connect between endpoints—consuming GiBs of DRAM [85, 88]. Shared Receive Queues (SRQ), like shRing, decrease memory use by sharing buffers. Whereas SRQ helps RDMA applicability by fitting I/O buffers in server DRAM, shRing improves performance by fitting I/O buffers in server LLC.

**Sharing Between Cores in Integrated NICs** Nebula [89] is an on-chip integrated NIC design optimized for RPC workloads. Nebula, like shRing, fits the I/O working set within the LLC. Whereas Nebula is applicable only for RDMA-like hardware-terminated protocols, shRing is applicable to typical general purpose Ethernet software network stacks.

## 8 Conclusions

Multicore systems with per-core Ethernet rings use too many receive rings, creating memory pressure that hampers performance. We show that shared receive rings alleviates this problem despite the associated synchronization costs.

## Acknowledgments

We thank the paper’s shepherd, Adam Belay, and the anonymous reviewers for their valuable feedback.

## References

- [1] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan, Daehoon Kim, Ren Wang, and Nam Sung Kim. IDIO: Network-driven, inbound network data orchestration on server processors. In *IEEE/ACM International Symposium on*

- Microarchitecture (MICRO)*, pages 480–493, 2022. <https://doi.org/10.1109/MICRO56248.2022.00042>.
- [2] Nambiar Amritha, Samudrala Sridhar, and Patil Kiran. Hardware acceleration of container networking interfaces. <https://legacy.netdevconf.info/0x14/session.html?talk-hardware-acceleration-of-container-networking-interfaces>, 2020. Accessed: 2022-10-10.
- [3] Amir Ancel, Tariq Tokun, and Saeed Mahameed. Rx and Tx bulking/batching. [https://legacy.netdevconf.info/2.1/slides/apr6/network-performance/04-amir-RX\\_and\\_TX\\_bulking\\_v2.pdf](https://legacy.netdevconf.info/2.1/slides/apr6/network-performance/04-amir-RX_and_TX_bulking_v2.pdf), 2017. Accessed: 2022-10-10.
- [4] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. Don't share, don't lock: Large-scale software connection tracking with krononat. In *USENIX Annual Technical Conference (ATC)*, pages 453–466, 2018. <https://www.usenix.org/conference/atc18/presentation/andre>.
- [5] ARM. ARM cache stashing. <https://developer.arm.com/documentation/102407/0100/Cache-stashing>, 2017. Accessed: 2022-12-10.
- [6] Dave Barach. VPP/software architecture. [https://wiki.fd.io/view/VPP/Software\\_Architecture](https://wiki.fd.io/view/VPP/Software_Architecture), 2018. Accessed: 2022-11-28.
- [7] Dotan Barak. `ibv_post_srq_recv`. [https://www.rdma Mojo.com/2013/02/08/ibv\\_post\\_srq\\_recv/](https://www.rdma Mojo.com/2013/02/08/ibv_post_srq_recv/). Accessed: 2022-09-26.
- [8] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5—16, 2015. <https://doi.org/10.1109/ANCS.2015.7110116>.
- [9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nikolai Zeldovich. An analysis of linux scalability to many cores. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, Vancouver, BC, October 2010. USENIX Association. <https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores>.
- [10] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, Internet Engineering Task Force, March 1999. <http://www.rfc-editor.org/rfc/rfc2544.txt>.
- [11] Jesse Brandeburg. `ice: change default number of receive descriptors`. <https://marc.info/?l=linux-netdev&m=156771568024262&w=2>, 2019. Intel. Accessed: June 2021.
- [12] Broadcom. NetXtreme E-Series PCIe NIC Ethernet Adapters Specification Sheet. <https://docs.broadcom.com/doc/netxtreme-e-series-pcie-nic-ethernet-adapters-specification-sheet>, 2021. Accessed: 2021-08-10.
- [13] Brad Bures, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. Intel's hyperscale-ready infrastructure processing unit (IPU). In *Hot Chips*, 2021. <https://doi.org/10.1109/HCS52781.2021.9567455>.
- [14] Idan Burstein. NVIDIA data center processing unit (DPU) architecture. In *Hot Chips*, 2021. <https://doi.org/10.1109/HCS52781.2021.9567066>.
- [15] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 65—77, 2021. <https://doi.org/10.1145/3452296.3472888>.
- [16] Cisco. TRex: Realistic Traffic Generator. <https://trex-tgn.cisco.com/>. (Accessed: May 2021.).
- [17] OASIS IDPF Technical Committee. IDPF (Infrastructure Data Path Function). [https://www.oasis-open.org/committees/download.php/70738/IDPF%20Spec\\_v0\\_9.pdf](https://www.oasis-open.org/committees/download.php/70738/IDPF%20Spec_v0_9.pdf), 2023. Accessed: 2023-05-13.
- [18] OASIS IDPF Technical Committee. OCP Server NIC SW Specification: Core Features. [https://docs.google.com/document/d/1FaVPGYipZ1sPhnYg7KItAS7ivL\\_svvZP8ZVJeFJezc0](https://docs.google.com/document/d/1FaVPGYipZ1sPhnYg7KItAS7ivL_svvZP8ZVJeFJezc0), 2023. Accessed: 2023-05-13.
- [19] Jonathan Corbet. Automatic buffer selection for `io_uring`. <https://lwn.net/Articles/815491/>, 2020. Accessed: 2023-04-13.
- [20] Intel Corporation. Intel data direct i/o technology (intel DDIO): A primer. <https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf>, 2012. Accessed: 2020-07-18.

- [21] Nithin Dabilpuram. [dpdk-dev] [patch 00/44] marvell CNXK ethdev driver. <https://inbox.dpdk.org/dev/20210306153404.10781-4-ndabilpuram@marvell.com/T>, 2021. Marvell. Accessed: 2022-11-28.
- [22] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of  $\mu$ -scale rpcs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 35–48, 2019. <https://doi.org/10.1145/3297858.3304070>.
- [23] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauch Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 373–387, 2018. <https://www.usenix.org/conference/nsdi18/presentation/dalton>.
- [24] Daly Dan. Introduction infrastructure programming. <https://ipdk.io/documentation/IPDK-io%20-%20Recipes.pdf>, 2021. Accessed: 2022-10-10.
- [25] Peter J. Denning. The working set model for program behavior. *Communications of the ACM (CACM)*, 11(5):323–333, May 1968. <https://doi.org/10.1145/363095.363141>.
- [26] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, 2009. <https://doi.org/10.1145/1629575.1629578>.
- [27] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.
- [28] Matt Faraclas. Received packets have been dropped by nic. <https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/>, 2014. Accessed: June 2021.
- [29] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *ACM Eurosys*, pages 1–17, 2019. <https://doi.org/10.1145/3302424.3303977>.
- [30] FreeBSD. Network RSS. <https://wiki.freebsd.org/NetworkRSS>, 2014. Accessed: January 2017.
- [31] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020. <https://www.usenix.org/conference/osdi20/presentation/fried>.
- [32] Fritz Kruger. CPU bandwidth - the worrisome 2020 trend. <https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/>, 2020. Accessed: 2021-06-09.
- [33] Or Gerlitz, Hadar Hen-Zion, Amir Vadai, and Rony Efraim. Introduction to switchdev SR-IOV offloads. [https://legacy.netdevconf.info/1.2/slides/oct6/04\\_gerlitz\\_efraim\\_introduction\\_to\\_switchdev\\_sriov\\_offloads.pdf](https://legacy.netdevconf.info/1.2/slides/oct6/04_gerlitz_efraim_introduction_to_switchdev_sriov_offloads.pdf), 2016. Accessed: 2022-10-10.
- [34] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 274–281, 2020. <https://doi.org/10.1145/3386367.3431295>.
- [35] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>.
- [36] 802.3-105 – IEEE standard for Ethernet. <https://doi.org/10.1109/IEEESTD.2016.7428776>, 2016.
- [37] InfiniBand Trade Association (IBTA). What is InfiniBand. <https://www.infinibandta.org/ibta-specification/>. (Accessed: Dec 2021).
- [38] Intel. Application Device Queues. <https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/adq-resource-center.html>. Accessed: 2022-09-29.

- [39] Intel. Processor Counter Monitor (PCM). <https://github.com/opcm/pcm>. Accessed: 2021-02-05.
- [40] Intel. Intel® 82544ei gigabit ethernet controller. <https://ark.intel.com/content/www/us/en/ark/products/2276/intel-82544ei-gigabit-ethernet-controller.html>, 2001. Accessed: 2022-10-10.
- [41] Intel. Intel® Xeon processors reach 2 gigahertz for workstations. <https://www.intel.com/pressroom/archive/releases/2001/20010925comp.htm>, 2001. Accessed: 2022-10-10.
- [42] Intel. Intel® 82598eb 10 gigabit ethernet controller. <https://ark.intel.com/content/www/us/en/ark/products/36918/intel-82598eb-10-gigabit-ethernet-controller.html>, 2007. Accessed: 2022-10-10.
- [43] Intel. Intel® Xeon® processor x5482. <https://ark.intel.com/content/www/us/en/ark/products/33088/intel-xeon-processor-x5482-12m-cache-3-20-ghz-1600-mhz-fsb.html>, 2007. Accessed: 2022-10-10.
- [44] Intel. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. <http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html>, Jan 2011.
- [45] Intel. Intel® Xeon® processor e7-2880 v2. <https://ark.intel.com/content/www/us/en/ark/products/75241/intel-xeon-processor-e72880-v2-37-5m-cache-2-50-ghz.html>, 2014. Accessed: 2022-10-10.
- [46] Intel. X710-am2. <https://ark.intel.com/content/www/us/en/ark/products/82944/intel-ethernet-controller-x710am2.html>, 2014. Accessed: 2022-10-10.
- [47] Intel. Tuning the buffers: a practical guide to reduce or avoid packet loss in dpdk applications. <https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/>, 2017. Accessed: June 2021.
- [48] Intel. Intel® Xeon® platinum 9282 processor. <https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html>, 2019. Accessed: 2022-10-10.
- [49] Intel. E810-cam1. <https://ark.intel.com/content/www/us/en/ark/products/187409/intel-ethernet-controller-e810cam1.html>, 2020. Accessed: 2022-10-10.
- [50] Intel. Intel ethernet network adapter e810-2cqda2. <https://ark.intel.com/content/www/us/en/ark/products/192561/intel-ethernet-network-adapter-e810-cqda1.html>, 2021. Accessed: 2021-08-10.
- [51] Intel Corporation. DPDK: Data plane development kit. <http://dpdk.org>, 2010. (Accessed: May 2016).
- [52] Intel Corporation. DPDK programmer’s guide: Poll mode driver. [https://doc.dpdk.org/guides/prog\\_guide/poll\\_mode\\_drv.html](https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html), 2014. (Accessed: Dec 2022).
- [53] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 567–584, 2022. <https://www.usenix.org/conference/nsdi22/presentation/iyer>.
- [54] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 3–14, 2013. <https://doi.org/10.1145/2486001.2486019>.
- [55] Zou Jia, Zhiyong Liang, and Yiqi Dai. Scalability evaluation and optimization of multi-core SIP proxy server. In *International Conference on Parallel Processing (ICPP)*, pages 43–50, 2008. [10.1109/ICPP.2008.30](https://doi.org/10.1109/ICPP.2008.30).
- [56] Rick A. Jones. Netperf: A network performance benchmark (Revision 2.0). <http://www.netperf.org/netperf/training/Netperf.html>, 1995. Accessed: August, 2016.
- [57] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for  $\mu$ second-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019. <https://www.usenix.org/conference/nsdi19/presentation/kaffes>.
- [58] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–186, 2018. <https://www.usenix.org/conference/nsdi18/presentation/katsikas>.



- [59] Amine Kherbouche. Scaleway natasha performance test. <https://github.com/scaleway/natasha/tree/master/test/perf>, 2018. Accessed: 2022-11-28.
- [60] Kevin Laatz. [dpdk-dev] [PATCH v2 0/3] Increase default RX/TX ring sizes. <https://mails.dpdk.org/archives/dev/2018-January/086889.html>, 2018. Intel DPDK. Accessed: June 2021.
- [61] Xueming Li. [dpdk-dev] [patch v11 0/7] ethdev: introduce shared rx queue. <https://lore.kernel.org/all/20211020075319.2397551-1-xuemingl@nvidia.com/>, 2021. Accessed: 2023-04-13.
- [62] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014. <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim>.
- [63] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 270—282, 2020. <https://doi.org/10.1145/3387514.3405868>.
- [64] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. DiskCryptlNet: Rethinking the stack for high-performance video streaming. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 211–224, 2017. <https://doi.org/10.1145/3098822.3098844>.
- [65] Marvell. FastLinQ 41000 Series Adapters. <https://www.marvell.com/content/dam/marvell/en/public-collateral/ethernet-adapter-andcontrollers/marvell-ethernet-adapter-fastlinq-41000-series-user-guide.pdf>, 2020. Accessed: June 2021.
- [66] Mellanox. Connectx@-6 en card product brief. [https://www.mellanox.com/sites/default/files/related-docs/prod\\_adapter\\_cards/PB\\_ConnectX-6\\_EN\\_Card.pdf](https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf), 2018. Accessed: 2019-08-06.
- [67] Mellanox. Mellanox NEO-Host. [https://www.mellanox.com/sites/default/files/related-docs/prod\\_management\\_software/PB\\_Mellanox\\_NEO\\_Host.pdf](https://www.mellanox.com/sites/default/files/related-docs/prod_management_software/PB_Mellanox_NEO_Host.pdf), 2018. Accessed: 2021-04-16.
- [68] Microsoft. Introduction to receive side scaling. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>, 2017. Accessed: January 2020.
- [69] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 217—231, 1999. <https://doi.org/10.1145/319151.319166>.
- [70] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 327—341, 2018. <https://doi.org/10.1145/3230543.3230560>.
- [71] NVIDIA. ConnectX@-7 Card Product Brief. <https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf>, 2021. Accessed: 2021-04-16.
- [72] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019. <https://www.usenix.org/conference/nsdi19/presentation/ousterhout>.
- [73] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 203–216, 2016. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda>.
- [74] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 207—218, 2013. <https://doi.org/10.1145/2486001.2486026>.
- [75] Solal Pirelli and George Candea. A simpler and faster NIC driver model for network functions. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2020.

<https://www.usenix.org/conference/osdi20/presentation/pirelli>.

- [76] Solal Pirelli, Akvilė Valentukonytė, Katerina Argyraki, and George Candea. Automated verification of network function binaries. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 585–600, 2022. <https://www.usenix.org/conference/nsdi22/presentation/pirelli>.
- [77] Jiri Pirko and Scott Feldman. Ethernet switch device driver model (switchdev). <https://www.kernel.org/doc/Documentation/networking/switchdev.txt>, 2015. Accessed: 2022-10-10.
- [78] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafir. Autonomous NIC offloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18—35, 2021. <https://doi.org/10.1145/3445814.3446732>.
- [79] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafir. The benefits of general purpose on-NIC memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1130—1147, 2022. <https://doi.org/10.1145/3503222.3507711>.
- [80] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325—341, 2017. <https://doi.org/10.1145/3132747.3132780>.
- [81] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to measure the killer microsecond. In *Proceedings of the Workshop on Kernel-Bypass Networks*, pages 37—42, 2017. <https://doi.org/10.1145/3098583.3098590>.
- [82] Scott Rixner. Network virtualization: Breaking the performance barrier: Shared I/O in virtualization platforms has come a long way, but performance concerns remain. *ACM Queue*, 6(1):36—44, January 2008. <https://doi.org/10.1145/1348583.1348592>.
- [83] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A High-Speed stateful packet processing approach for tbps programmable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1237–1255, 2023. <https://www.usenix.org/conference/nsdi23/presentation/scazzariello>.
- [84] Jeff Shafer, David Carr, Aravind Menon, Scott Rixner, Alan Cox, Willy Zwaenepoel, and Paul Willman. Concurrent direct network access for virtual machine monitors. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 306–317, 01 2007. <https://doi.org/10.1109/HPCA.2007.346208>.
- [85] Galen M Shipman, Timothy S Woodall, Richard L Graham, Arthur B Maccabe, and Patrick G Bridges. Infiniband scalability in open MPI. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. <https://doi.org/10.1109/IPDPS.2006.1639335>.
- [86] Shahaf Shuler. [dpdk-dev] [patch v2 2/2] net/mlx5: add rx and tx tuning parameters. <https://mails.dpdk.org/archives/dev/2018-May/099834.html>, 2018. Mellanox. Accessed: Nov. 2022.
- [87] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafir. Ioctopus: Outsmarting nonuniform dma. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 101–115, 2020. <https://doi.org/10.1145/3373376.3378509>.
- [88] S. Sur, Lei Chai, Hyun-Wook Jin, and D.K. Panda. Shared receive queue based scalable MPI design for InfiniBand clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. <https://doi.org/10.1109/IPDPS.2006.1639336>.
- [89] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The nebula rpc-optimized architecture. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020. <https://doi.org/10.1109/ISCA45697.2020.00027>.
- [90] Herbert Tom and de Bruijn Willem. Scaling in the linux networking stack. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>, 2011. Accessed: 2020-03-05.
- [91] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283—297, 2018. <https://www.usenix.org/conference/nsdi18/presentation/tootoonchian>.

- [92] Tariq Toukan. [PATCH net-next 08/10] net/mlx4\_en: Increase default TX ring size. <https://www.mail-archive.com/netdev@vger.kernel.org/msg173779.html>, 2017. Mellanox. Accessed: June 2021.
- [93] S. Van Doren. Compute express link. In *IEEE Symposium on High Performance Interconnects (HOTI)*, 2019. <https://doi.org/10.1109/HOTI.2019.00017>.
- [94] VMware. Large packet loss in the guest os using vmxnet3 in esxi (2039495). <https://kb.vmware.com/s/article/2039495>, 2021. Accessed: June 2021.
- [95] Ziyue Yang, Ben Walker, James R Harris, Yadong Li, and Gang Cao. Optimal use of the tcp/ip stack in user-space storage applications with ADQ feature in NIC. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 363–371, 2020. <https://doi.org/10.1109/ICPADS51040.2020.00056>.
- [96] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Iliia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the I/O when allocating your LLC. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021. <https://doi.org/10.1109/ISCA52012.2021.00018>.
- [97] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 141—154, 2017. <https://doi.org/10.1145/3098822.3098833>.

# ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta

Harshit Saokar<sup>1</sup>, Soteris Demetriou<sup>1,2</sup>, Nick Magerko<sup>1</sup>, Max Kontorovich<sup>1</sup>, Josh Kirstein<sup>1</sup>, Margot Leibold<sup>1</sup>, Dimitrios Skarlatos<sup>1,3</sup>, Hitesh Khandelwal<sup>1</sup>, and Chunqiang Tang<sup>1</sup>

<sup>1</sup> Meta Platforms, <sup>2</sup> Imperial College London, <sup>3</sup> Carnegie Mellon University

## Abstract

Datacenter applications are often structured as many interconnected microservices, and the service mesh has become a popular approach to route RPC traffic among services. This paper presents *ServiceRouter* (SR), Meta’s global service mesh, which has been in production since 2012. SR differs from publicly known service meshes in several important ways. First, SR is designed for hyperscale and currently uses millions of L7 routers to route tens of billions of requests per second across tens of thousands of services. Second, while SR adopts the common approach of using sidecar or remote proxies to route 1% of RPC requests in our fleet, it employs a routing library that is directly linked into service executables to route the remaining 99% directly from clients to servers, without the extra hop of going through a proxy. This approach significantly reduces the hardware costs of our hyperscale service mesh, saving hundreds of thousands of machines. Third, SR provides built-in support for sharded services, which account for 68% of RPC requests in our fleet, whereas existing general-purpose service meshes do not support sharded services. Finally, SR introduces the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed datacenter regions, which, to our knowledge, has not been attempted before.

## 1 Introduction

The increasing need for continuous integration and deployment [25] in datacenter environments has led to the widespread adoption of the microservice architecture [?, 42], in which an application is decomposed into a collection of services that can be independently developed and deployed. To manage the traffic of remote procedure calls (RPCs) between these services, many organizations use a service mesh [30].

Figure 1 shows the most common form of layer-7 (L7, i.e., application layer) service mesh. In this architecture, each service process is accompanied by an L7 sidecar proxy running on the same machine, which routes RPC requests on behalf of the service. As an example, when service A on machine 1 sends requests to service B, the proxy on machine 1 will load-balance the requests across machines 2 and 3. If the

autoscaling system detects an increase in load and starts a new replica of service B on machine 4, the control plane’s service discovery function will notify the proxy on machine 1, which will then include machine 4 in its load-balancing targets for future requests for service B.

This paper presents Meta’s global service mesh called *ServiceRouter* (SR). SR supports a comprehensive set of features, including service discovery, load balancing, failover, authentication, encryption, observability [1], overload protection [39], distributed request tracing [32], resource attribution for capacity management [16], and duplication of traffic for shadow testing. Due to space limitations, the focus of this paper is primarily on answering the following questions: (1) how to scale a service mesh to millions of L7 routers, (2) how to minimize the hardware cost of a hyperscale service mesh, (3) how to support sharded services which are essential but often overlooked, and (4) how to simultaneously minimize RPC latency and balance load in a geo-distributed service mesh.

**Scalability.** Traditionally, a software-defined network [18] uses a centralized control plane and a decentralized data plane. Most service meshes [10,30,37] follow this approach and use a central controller to configure the routing table of each sidecar proxy. However, this approach is not sufficiently scalable for a hyperscale service mesh. The control plane has a dual function of generating global routing metadata and managing each L7 router. We advocate for keeping the former in the central control plane, but decentralizing the latter by transferring its function to L7 routers. Each L7 router should be self-configuring and self-managing so that the central control plane can scale out easily.

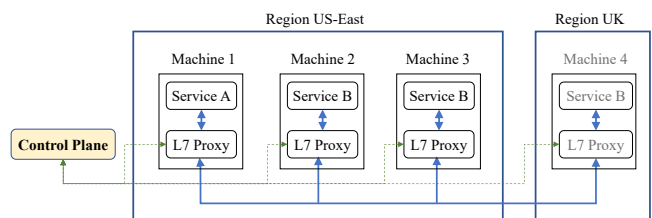


Figure 1: Sidecar-proxy-based service mesh.



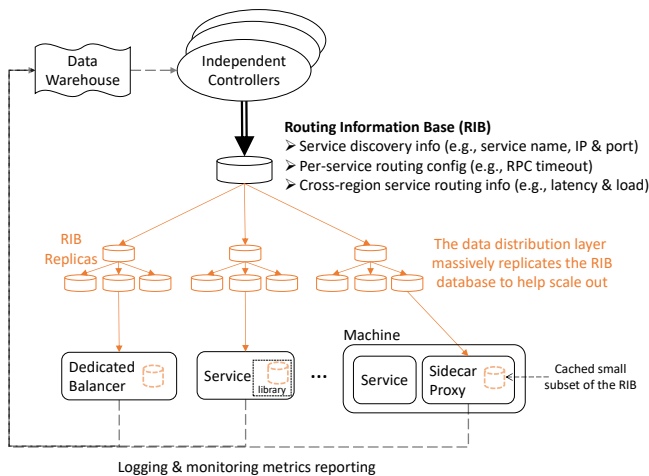


Figure 2: ServiceRouter’s scalable service-mesh architecture.

Figure 2 shows SR’s scalable architecture. On the top, different controllers independently execute functions such as registering services and generating a per-service cross-region routing table. Each controller independently updates the central *Routing Information Base* (RIB), and is not concerned with configuring or managing individual L7 routers. In the middle of Figure 2, the distribution layer replicates the RIB so that there are sufficient RIB replicas to handle read traffic from millions of L7 routers. At the bottom, guided by the RIB, each L7 router self-configures without the control plane’s direct involvement. Initially, an L7 router’s routing table is empty. When it receives an RPC request that targets a service, it fetches the routing information for the service from an RIB replica and subscribes to future RIB updates for the service.

**Hardware cost.** Existing service meshes [10, 30, 37] use sidecar proxies to forward requests (Figure 1). However, this approach incurs extra hardware costs due to the overhead of the extra routing hop, such as data serialization and deserialization in the proxy. Istio’s benchmarking [47] shows that 0.35 vCPU can handle 1,000 requests per second. Therefore, it would need the equivalent of 1,750,000 AWS `t4g.small` VMs to route 10 billion requests per second.

SR eliminates the need for a proxy and its associated hardware cost by providing the service-mesh function through a library called *SRLib*. *SRLib* is linked into service executables and routes RPC requests directly from clients to servers. However, this approach requires changes to services’ source code, which is not always possible. For example, our services written in Erlang cannot link *SRLib* into their executables.

To meet the diverse needs of services, SR enables the seamless coexistence of different types of L7 routers, including Istio-style sidecar proxies, AWS-ELB-style [5] dedicated load balancers, and gRPC-style lookaside [24] load balancers, as shown in Figure 2. The key insight that enables SR’s versatility is that the controllers at the top of Figure 2 are agnostic to the L7 routers at the bottom, allowing the L7 routers to choose their own architecture.

The embedded *SRLib* helps us achieve significant hardware savings. Currently, 99% of RPC requests at Meta are routed by *SRLib*, and the remaining 1% is routed by sidecar proxies and a group of dedicated load balancers that consume thousands of machines. If we were to completely switch from using *SRLib* to using proxies to route 100% of our traffic, we would need to add hundreds of thousands of extra machines.

**Sharded services.** Sharding [34] and replication are two key techniques for building scalable services. In our fleet, the vast majority of RPC traffic is for sharded services. Despite their importance, existing general-purpose service meshes do not directly support routing for sharded services. For example, in Figure 1, assuming that Service B’s replicas on machines 2, 3, and 4 host various data shards that can dynamically migrate across machines, it is possible for the proxy on machine 1 to route a request to machine 2 mistakenly, even if the request is meant for a shard on machine 3.

SR makes sharding support a top priority and uses a single framework to support both sharding and replication. As sharding is often tied to application logic, our key insight is to enforce separation of concerns by defining a simple and generic sharding abstraction between the service mesh and services. This allows SR to route traffic for different sharded services without needing to know their application logic.

**Cross-region routing.** Existing solutions [4, 41] are not optimized for routing across geo-distributed datacenter regions. For example, in Figure 1, should machine 1 route requests to machines 2 and 3, which have a higher load, or to machine 4, which has a longer network latency? Moreover, how to ensure that the resulting global traffic distribution for a service matches the global supply of the service’s capacity in different regions? These questions have not been well answered before.

To better support cross-region routing, we introduce the concept of locality rings for services to express their preferred tradeoff between latency and load. For example, a service can instruct SR that if and only if the load in the local region goes above 70%, SR can relax the locality constraint and route some local traffic to other regions in the same continent; if the load further increases above 80%, SR can even route some local traffic to regions in a different continent. SR collects global traffic and load information for each service, computes a cross-region routing table that conforms to the requirements specified in locality rings, and disseminates the routing table to L7 routers to guide their routing. This allows SR to provide globally optimized traffic shaping for services.

**Contributions.** We summarize our contributions below.

- SR is designed for hyperscale. While there may be proprietary systems of a similar scale, their specifics are not publicly available, and existing open-source service meshes do not scale well [57]. We hope that our experience can be helpful to those who seek to build scalable service meshes.

- SR supports the seamless coexistence of different types of L7 routers in one service mesh, including sidecar proxies [30], dedicated load balancers [5], lookaside load balancers [24], and an embedded routing library. To save on hardware costs, SR routes 99% of RPC requests in our fleet using the embedded library. This approach, along with the scale at which we utilize it, might be unique in the industry.
- While existing service meshes exclusively focus on unsharded services, which only account for 32% of our fleet’s RPC requests, SR provides built-in support for sharded services, which account for 68% of our traffic.
- Although primitive forms of locality-aware routing existed before [31], our novelty is to introduce the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed regions.

## 2 Comparison of Services Mesh Architectures

In this section, we compare different architectures of service mesh. The design space, shown in Table 1, is determined by the answers to two key questions: (1) which component fetches and caches the routing metadata, and (2) which component routes application RPC traffic. In Table 1, *Library*, *Kernel*, *Local*, and *Remote* mean that RPC routing or maintenance of routing metadata is performed by an embedded library, the kernel, a local proxy/daemon on the RPC client machine, or a remote proxy/service outside the client machine, respectively.

### 2.1 Different Types of L7 Routers in SR

SR allows different L7 router setups to coexist in one service mesh in order to support diverse use cases. These setups are shown in Figure 4 and explained below. Different types of L7 routers interoperate well and can send RPC requests to the same server at the same time.

**SRLib.** This setup is shown in Figure 4(a) and corresponds to solution (9) in Table 1. It provides the service-mesh function through a library, which is directly linked into the RPC client’s executable. The library can route requests directly to servers without the need for a proxy, eliminating the extra hardware cost and routing latency of a proxy. The client only needs to fetch and cache a small part of RIB (called the *miniRIB*) that is actively used by the client.

We run a separate *RIBDaemon* on the client machine to cache *miniRIB*, instead of relying on SRLib to do so. This

		Which component manages and caches <i>miniRIB</i> ?			
		Lib	Kernel	Local	Remote
Which component forwards application RPC traffic?	Lib	(1) ✗	(5) ✗	(9) SRLib	(13) SRLookaside
	Kernel	(2) ✗	(6) eBPF	(10) ✗	(14) ✗
	Local	(3) ✗	(7) ✗	(11) SRSidecarProxy	(15) SRSidecarProxy plus SRLookaside
	Remote	(4) ✗	(8) ✗	(12) ✗	(16) SRRemoteProxy

Table 1: The complete solution space for service mesh. The symbol ✗ indicates undesirable solutions.

separation allows for the use of *cgroup* to provide strong isolation between a) *RIBDaemon*’s less urgent background work that keeps *miniRIB* up-to-date and b) *SRLib*’s latency-sensitive foreground work that routes RPC requests and is on the critical path of application performance. Updates to RIB can be very spiky and when those updates are pushed to *miniRIB*, they can cause a spike in CPU usage to process the updates. Figure 3 shows the spiky CPU usage of a production machine’s *RIBDaemon*. When *cgroup* throttles *RIBDaemon*, it has little impact on *SRLib* because *SRLib* consults *RIBDaemon* only once on its first RPC for a service and all subsequent RPCs for the service go directly from *SRLib* to servers without involving *RIBDaemon*. In contrast, if *miniRIB* is managed by *SRLib*, *cgroup* cannot isolate the resource usage for maintaining *miniRIB* from the application’s own resource consumption because *SRLib* is linked into the application.

**SRLookaside.** This setup, shown in Figure 4(b) and corresponding to solution (13) in Table 1, addresses the issue of *RIBDaemon* running on every RPC client machine and consuming resources, particularly memory. It eliminates *RIBDaemon* by moving the function of *miniRIB* management and server selection to a remote and shared *SRLookasideService*, while still routing RPCs directly from clients to servers without going through an intermediate proxy.

Historically, Meta used a large fleet of small machines with as little as 16GB memory because of their advantages in power efficiency. Accordingly, *SRLookaside* was developed to save memory on those small machines. Now even our small machines have at least 64GB memory and hence the usage of *SRLookaside* was deprecated, because the limited memory savings no longer justify the burden of maintaining the *SRLookaside* service.

**SRSidecarProxy.** This setup, shown in Figure 4(c) and corresponding to solution (11) in Table 1, incurs extra hardware costs and routing latency like Istio [30], but its implementation is much more scalable than Istio, because each *SRProxy* self-manages without the control plane’s involvement and only caches *miniRIB* instead of the entire RIB. At Meta, the usage of *SRSidecarProxy* is mostly limited to services written in Erlang because *SRLib* does not directly support Erlang.

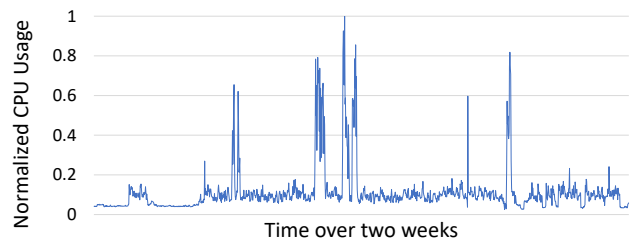


Figure 3: Spiky CPU usage of a machine’s *RIBDaemon*.

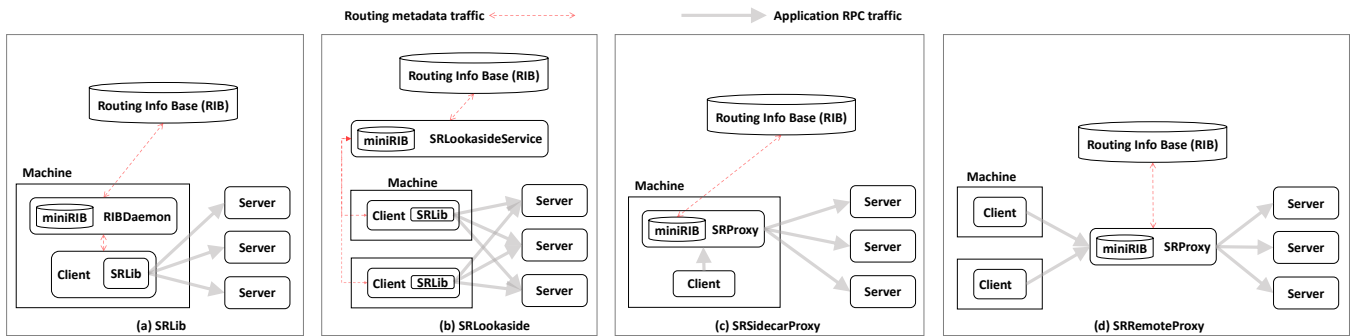


Figure 4: Service mesh design alternatives. The diagrams show how RPC clients send requests to RPC servers.

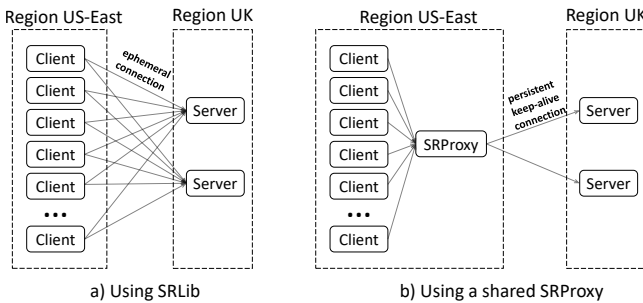


Figure 5: A shared SRProxy is better at dealing with many clients sending infrequent cross-region RPC requests.

**SRRemoteProxy.** This setup, shown in Figure 4(d) and corresponding to solution (16) in Table 1, is similar to AWS ELB [5]. SRRemoteProxy functions as a dedicated load balancer shared by multiple clients, reducing the number of RPC connections and increasing the reuse of keep-alive connections, as illustrated in Figure 5. Suppose there are a large number of clients and each client sends a request to a server in a remote datacenter region occasionally. Each RPC would experience a long delay due to the three rounds of cross-region round-trip time needed to establish a new TLS/TCP connection. A shared proxy eliminates this overhead by keeping a small number of cross-region connections alive and reusing them to send requests on behalf of many clients.

## 2.2 Comparison of L7 Routers

Next, we compare solutions in Table 1. Solutions (1)–(4) are undesirable because managing miniRIB in the library would impact the application’s performance due to lack of isolation. Solutions (5), (7), and (8) are undesirable because there is no system call to access miniRIB cached in the kernel. Although solution (6) exists in the form of eBPF-based service mesh [35], its function is limited by what can be done by an eBPF program in the kernel. For example, Cilium [29]’s eBPF program can only handle L3/L4 protocols and it still has to use a sidecar proxy to handle L7 protocols. Similar to solution (6), solutions (10) and (14) are undesirable because of the difficulty of implementing advanced L7 routing features in the kernel.

Service Mesh Alternatives	A1: HW cost	A2: direct RPC	A3: fast RIB	A4: save mem	A5: unchg code	A6: share conn
SRLib	✓	✓	✓	≈	✗	✗
Sidecar Proxy	✗	✗	✓	≈	✓	✗
Remote Proxy	✗	✗	✓	✓	✓	✓
Lookaside	≈	✓	✗	✓	✗	✗

Attributes	Description
A1: HW cost	No extra hardware cost for proxy or lookaside service.
A2: direct RPC	Application RPC traffic goes from clients to servers without the overhead of going through an intermediate proxy.
A3: fast RIB	No overhead to access Routing Information Base (RIB) outside the client machine thanks to local RIB caching.
A4: save mem	No extra memory usage on the client machine thanks to the elimination of the local RIB cache.
A5: unchg code	No need for application source code modification.
A6: share conn	Benefits of multiple clients sharing a proxy, e.g., better load balancing or connection reuse (Figure 5).

Alter-natives	When to use a particular service-mesh setup	Usage at Meta
SRLib	Use SRLib for large-scale deployments where hardware costs and routing latency are most important.	99% of traffic
Remote Proxy	Use remote proxies if it benefits from multiple clients sharing a proxy, e.g., to improve connection reuse when there are many low-traffic clients (Figure 5).	Some limited use
Sidecar Proxy	Use sidecar proxies if you cannot modify application source code to use SRLib, or SRLib does not support the app’s programming language (e.g., Erlang).	Only one-off use
Lookaside	Use a remote lookaside service to reduce the memory used on every client machine for caching miniRIB.	Depreciated

Table 2: Comparison of service mesh design alternatives.

Solution (12) is undesirable because it is strictly worse than (16), i.e., if routing is performed by a remote proxy, it is better to move miniRIB to the remote proxy as well. Theoretically, solution (15) uses less memory on the client machine than (11) does. However, (15) is not used at Meta since even (11) is not widely used and the added benefit of (15) is limited.

Finally, for ease of access, we summarize in Table 2 the comparison of the design alternatives.

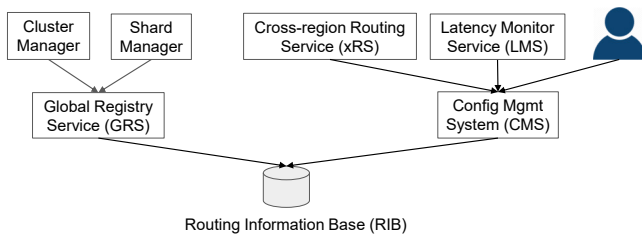


Figure 6: ServiceRouter’s control-plane components.

### 3 ServiceRouter Design

In this section, we first present an overview of ServiceRouter and then elaborate on its key design ideas.

#### 3.1 Overview

SR supports all four types of L7 routers depicted in Figure 4. For the sidecar or remote proxy setup, we add a wrapper layer atop SRLib code to run it as a standalone proxy. SR’s control-plane components are depicted in Figure 6 and further explained below.

**Routing Information Base (RIB).** RIB is a Paxos-based key-value store with nine Paxos acceptors distributed across five geographic regions to ensure high availability. It centrally stores routing metadata for all services running in all regions. It uses thousands of Paxos learners to create many local RIB replicas in every region to ensure high read throughput and availability even if a region is disconnected from other regions. We discuss how to scale RIB in §4.1.

**Global Registry Service (GRS).** GRS maintains service and shard discovery information in RIB. Figure 7 shows two example services registered at GRS. *Service A* is replicated but not sharded. When the cluster manager [53] starts or stops a container for *service A*, it informs GRS to update the list of *service A*’s replicas. We will explain SR’s built-in support for sharded services in §3.3.

**Configuration Management System (CMS).** CMS [52] allows customization of the routing policy for each service, including RPC timeout, connection reuse, locality routing preference, etc. Services owners follow the configuration as code paradigm to author, review, and commit routing configurations. It also supports automated configuration updates. For example, the latency monitoring service (LMS) periodically aggregates and commits configuration updates related to cross-region latency to guide SRLib’s routing decisions.

**Cross-region Routing Service (xRS).** Compared with a centralized load balancer, SRLib only has a local view of the traffic from one client and might not make globally optimal routing decisions. xRS addresses this problem by aggregating global traffic information for each service and computing a cross-region routing table, which is disseminated via RIB and consumed by SRLib to guide its routing decisions.

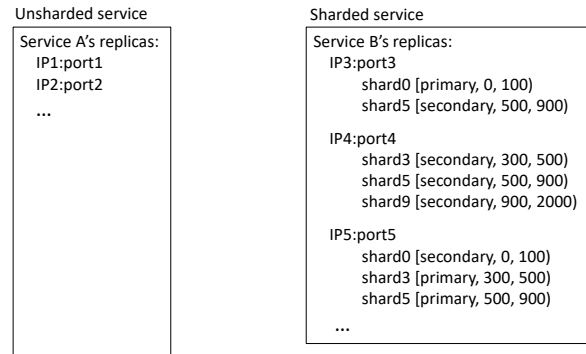


Figure 7: Examples of GRS’ service registry records.

#### 3.2 Service Discovery

A RIBDaemon runs on each machine and maintains a so-called miniRIB that caches the specific parts of RIB that are needed by the RPC clients running on the machine. Initially, miniRIB is empty. When SRLib wants to send an RPC request to a particular service, such as *service X*, it requests *service X*’s routing metadata from RIBDaemon. RIBDaemon fetches the metadata from a RIB replica, caches it on disk so that it can survive machine reboots, subscribes to future updates related to *service X*, and finally returns the metadata to SRLib. SRLib also subscribes to RIBDaemon for future updates and caches the metadata in memory (but not on disk) for later reuse so that it won’t contact RIBDaemon on every RPC request.

When the deployment of *service X* is changed in the future, the cluster manager informs GRS to update RIB. The update is immediately pushed to all RIB replicas, which further push the update to every RIBDaemon that subscribes to *service X*’s routing metadata. Finally, RIBDaemon pushes the update to SRLib. *Service X* may be deployed in multiple datacenter regions, and its replicas in each region are managed by a different regional cluster manager. All of these cluster managers inform GRS to update the same service-registry record for *service X* so that a client’s RPC request can potentially be routed to a replica in any region (§3.4.1). The RPC client of a service can choose to send requests only to servers located in the same region as the client. In this scenario, to reduce overhead, RIBDaemon subscribes only to routing updates originating from the local region.

With the help of the cluster manager, clients do not need to independently discover a server’s failure through timeouts. When a server is brought down for planned maintenance, such as code deployment, the cluster manager first updates RIB to inform the clients and then stops the server. For unplanned failures, the cluster manager detects all kinds of failures, such as process crashes/hangs and machine failures, and updates RIB to inform the clients.

#### 3.3 Support for Sharded Services

SR provides built-in support for sharded services. In Figure 7, *service B* is both sharded and replicated. We define a sim-



ple sharding abstraction between SR and services to enforce separation of concerns, so that SR can route traffic without needing to know a sharded service’s internal application logic. Specifically, a service specifies how a 128-bit key space is partitioned into shards. Each shard can be independently replicated and migrated across containers. Each shard replica is associated with an abstract role, e.g., *primary* or *secondary*. In this example, *shard5* corresponds to the key range [500, 900) and its replica on *IP4:port4* serves the *secondary* role. SR is not concerned with the real semantics of the shard key or role, and merely routes requests according to a client’s request.

```
SRClient *cln = SR_get_client("ServiceB", 618/*key*/, SECONDARY);
cln->foo(); // Invoke RPC for foo().
```

In this example, SR discovers that *shard5* contains key 618 and *shard5*’s *secondary* role is served by its replicas on *IP3:port3* and *IP4:port4*. SR picks one of them to serve the request according to the load balancing policy. In the service’s implementation, the *primary* and *secondary* roles could be mapped to the leader and follower replicas of a database, respectively.

SR’s shard-map abstraction is generic and currently supports hundreds of sharded services. Most but not all of them are managed by a common shard manager [34], which notifies GRS to update the shard registry when new shards are added or removed, or existing shards are migrated across containers.

With SR, sharded and unsharded services share and reuse all the sophisticated components in SR (Figures 2 and 6). Moreover, routing for sharded services works out-of-the-box without any additional effort. In contrast, existing general-purpose service meshes do not support sharded services, and applications have to develop their own solutions.

**Design alternatives.** One alternative to SR’s shard-map approach is *consistent hashing* [33]. Given a list of servers, it deterministically determines the server responsible for a given key based on hashing. As a result, it does not need to store the *shard map* in Figure 7. Despite its advantage in simplicity, consistent hashing is insufficient for advanced sharding use cases, as its deterministic key assignment does not support dynamic migration of shards in response to shard load changes [2, 34]. SR provides built-in support for both consistent hashing and the shard-map approach. As shown in our previous work [34], out of the hundreds of sharded services at Meta, the number of services that choose to use a flexible shard map is 5.4 times higher than the number of services that choose to use consistent hashing, which confirms the importance and effectiveness of the shard-map approach.

Another alternative to SR’s shard-map approach is to allow a service to provide its own custom lookaside-service implementation. This approach can provide maximum flexibility and separate the service’s custom shard discovery and selection logic from the service mesh. Both gRPC [24] and SR’s lookaside interfaces can support this approach. At Meta, some service owners were initially interested in this approach

because of its flexibility. However, they ultimately did not use it because of the burden of maintaining a custom lookaside service, and also because it turns out that the shard-map approach and consistent hashing together are sufficient for nearly all sharded services.

### 3.4 Load Balancing

SR’s load-balancing solution is based on the `Pick-2` [41] algorithm. `Pick-2` randomly samples two servers from a candidate pool and chooses the server with less load as the RPC target. However, using `Pick-2` alone is not sufficient for a geo-distributed service mesh. Therefore, we have developed three novel techniques to complement `Pick-2`: 1) Consider regional locality when sampling two random servers (§3.4.1). 2) Sample two random servers from a stable subset of servers, rather than all servers, to maximize connection reuse (§3.4.2). 3) Take an adaptive approach to load estimation based on the workload characteristics (§3.4.3). Further details on these techniques are provided in the following sections.

#### 3.4.1 Locality Awareness

In a geo-distributed service mesh, a faithful implementation of `Pick-2` would cause long RPC latencies because it does not take regional locality into account. Our measurements show that the P50 of within-region RTT is only 116 $\mu$ s, while the P50 of cross-region RTT is 35ms and the P99 is as high as 163ms. These data emphasize the importance of considering regional locality when routing RPC requests.

Instead of `Pick-2`’s approach of randomly sampling two servers from the candidate pool, SR uses the so-called *locality rings* to filter out long-latency servers that are far from the client, and then sample from the remaining nearby servers. Each service can define a set of rings with increasing latencies, e.g., [ring<sub>1</sub>: 5ms | ring<sub>2</sub>: 35ms | ring<sub>3</sub>: 80ms | ring<sub>4</sub>:  $\infty$ ]. The Latency Monitoring Service (LMS) periodically updates RTTs between regions, and RPC clients obtain them via CMS.

An RPC client uses cross-region RTTs to estimate its latency to different servers. Starting from ring<sub>1</sub>, if the client finds any RPC server whose latency is within the latency bound for ring<sub>*i*</sub>, it filters out all servers in ring<sub>*i*+1</sub> and above, and randomly samples two servers from ring<sub>*i*</sub>. If the service has no servers in ring<sub>*i*</sub>, it considers servers in ring<sub>*i*+1</sub>, and so forth. SR’s default setting maps [ring<sub>1</sub>|ring<sub>2</sub>|ring<sub>3</sub>|ring<sub>4</sub>] to [same region | neighboring regions | same continent | global].

Filtering by locality rings reduces routing latencies but still has limitations due to lack of a global view. First, servers in ring<sub>*i*</sub> might be overloaded while servers in ring<sub>*i*+1</sub> are underutilized. Second, clients’ local routing decisions might not lead to an optimal global traffic distribution that matches the global supply of server capacity. In particular, when a region *X* fails, if all clients independently decide to reroute their requests initially going to *X*, to *X*’s nearest region *Y*, they may overload *Y*, bring it down, and together move onto the next region *Z*, and so forth, causing a domino effect.

The Cross-region Routing Service (xRS) solves these problems by using global information to compute a per-service cross-region routing table whose entry  $[P_{ij}]$  means that  $P_{ij}$  fraction of the service's RPC requests originated from region  $R_i$  should be routed to region  $R_j$ . The cross-region routing table is stored in RIB and disseminated to all clients. When an RPC client wants to send a request, it follows the traffic distribution  $P_{ij}$  to randomly choose a destination region, and then applies the normal routing algorithm to select a server in the destination region.

xRS can purposely update the routing table to shift traffic out of a region in preparation for an upcoming maintenance event or in response to a disaster. While doing so, it tries to avoid overloading other regions. Guided by a PID controller [50], it gracefully shifts traffic across regions to prevent over-reaction. If there is insufficient capacity globally, it creates so-called black holes in the routing table to instruct clients to drop certain traffic instead of overloading servers.

Next, we describe how xRS computes the cross-region routing table. xRS collects traffic and load information globally, and simulates how a region's load would change if more or less traffic is routed to the region. For each service, xRS periodically fetches load information from its servers and aggregates it by region. It also collects requests per second (RPS) served by servers in each region, which is used to calculate the *RPS cost* as the ratio of a region's load to its RPS. *RPS cost* is the estimated load increase due to a unit of RPS increase. For example, the load for a region with a 60% load serving 100 RPS, would increase by 0.6% if 1 RPS is added to the region.

xRS strives to simultaneously minimize RPC latency and balance load across regions. It expands the locality rings with load thresholds, e.g.,  $[\text{ring}_1: 5ms : 55\% \mid \text{ring}_2: 35ms : 65\% \mid \text{ring}_3: 80ms : 80\% \mid \text{ring}_4: \infty : \infty]$ . Intuitively, it means that, for example, when  $\text{ring}_1$ 's load goes beyond 55%, xRS will relax its latency restriction and start to consider routing traffic to servers in  $\text{ring}_2$ , and so forth. This load-enriched locality ring information is not directly consumed by SRLib, but instead is fed to xRS to compute a per-service cross-region routing table as follows. xRS first tries to serve all requests in the source region locally, by setting  $\forall i P_{ii} = 1$  and  $\forall i \forall j \neq i P_{ij} = 0$ . Then assisted by each region's *RPS cost*, it identifies the most loaded region and tries to follow the preference in the locality rings to move some of the region's traffic to nearby regions. This process repeats until either no regions are overloaded or all regions are equally loaded.

Currently, 46% of our services are routed using xRS' cross-region routing tables, while the rest are routed using the baseline locality rings without the routing tables. Some services choose not to use xRS due to the overhead of collecting global traffic and load information. Moreover, some services generate high traffic and require low latency, and as a result, they prefer to fail a request instead of routing it across regions.

In total, about 16% of RPC requests in our fleet are routed across regions. This emphasizes the importance for global

service meshes to optimize cross-region routing, an area that is largely overlooked by existing service meshes.

**Design alternative.** With xRS, service owners need to apply their domain knowledge to set the thresholds for network RTT and server utilization in locality rings. To avoid the burden of setting these thresholds, an alternative approach is to use end-to-end RPC latency as the sole metric, which, in theory, would automatically consider both network RTT and server utilization. The load-balancing goal of this latency-focused approach would be to minimize the average RPC latency. Pacifici et al. [45] used a similar approach in a local cluster setting. However, SR does not follow this approach because, based on both queuing theory [11] and our production experience, modeling latency at high utilization is not robust. This implies that xRS would not be able to accurately predict how traffic shifts would affect RPC latency.

Moreover, minimizing RPC latency by trading long queuing delay at the RPC server for long cross-region network RTT is not a robust method, as it can lead to overloading of nearby servers and a high RPC error rate. We explain this through an example. Suppose a client sends requests to two servers  $X$  and  $Y$ , where  $X$  is in the same region with a  $100\mu s$  RTT and  $Y$  is in a different region with a  $100ms$  RTT. Further assume that it takes  $1ms$  to process a request. To minimize the RPC latency, the latency-focused approach would send all requests to  $X$ , which is in the local region, until its queuing delay reaches  $100ms$ , and only then it would start to send requests to  $Y$ , which is in a remote region. However, with a processing time of  $1ms$ , when the queuing delay at  $X$  reaches  $100ms$ ,  $X$  would be severely overloaded and might experience a high error rate. Overall, in a geo-distributed environment where network RTT may vary by three orders of magnitude, from  $100\mu s$  to  $100ms$ , the latency-focused approach is not robust.

### 3.4.2 RPC Connection Reuse

Our measurements show that setting up a new TLS/TCP connection takes  $1.6ms$  and consumes  $14KB$  of memory on each side. To reduce this overhead, SR keeps the TLS/TCP connections and reuses them across different RPC requests. However, the randomization used by `Pick-2` makes connection reuse ineffective. As `Pick-2` randomly samples two servers out of all  $n$  servers for each request, over time, an RPC client communicates with all  $n$  servers. However, it is impractical to maintain keep-alive connections with all  $n$  servers when  $n$  is large because of the memory and CPU overhead required to maintain the connections.

To improve connection reuse, an RPC client chooses a stable subset of  $k$  servers out of all  $n$  servers (often  $k \ll n$ ), and keeps reusing these  $k$  stable servers. Upon each RPC request, `Pick-2` samples two servers out of the  $k$  stable servers instead of all  $n$  servers. Over time, the client maintains keep-alive connections with the  $k$  stable servers.

One challenge is for each RPC client to independently choose their  $k$  stable servers while globally the load spreads

evenly across all  $n$  servers. Suppose a server on average maintains keep-alive connections with  $M$  clients. When a new server is added to the existing  $n$  servers, an ideal and stable solution should require only  $M$  clients to drop one existing server out of their list of  $k$  stable servers and add the new server to their list, so that the new server also serves  $M$  clients like other servers.

With SR, each RPC client uses Rendezvous Hashing [9,54] to select  $k$  stable servers, which achieves the ideal properties described above. Specifically, a client uses its unique client ID as hashing salt, computes the hashcodes of all servers, and chooses the  $k$  servers with the largest hashcode. Stable servers and Rendezvous Hashing together help SR maximize connection reuse. In production, over 99% of our RPC requests reuse existing connections.

**Design alternative.** We prefer Rendezvous Hashing over Consistent Hashing [33] because it allows SR to use weighted hashing to assign client connections proportional to a server’s compute power. This in combination with a weighting mechanism to bias the `Pick-2` probability proportional to a server’s compute power, solves the problem that our large fleet runs multiple generations of hardware that have very different performance characteristics. Moreover, Rendezvous Hashing achieves better load balancing. For example, when a server dies, its load is evenly redistributed to other servers even without using Consistent Hashing’s virtual servers.

### 3.4.3 Adaptive Load Estimation

In order for `Pick-2` to choose a routing target between two candidates, it needs to know the load information. By default, SR uses the number of outstanding requests at an RPC server to represent its load. In addition, SR allows custom load metrics such as CPU, memory, disk, or any application-level metric. Currently, 77% and 18% of the RPC requests in our fleet use the number of outstanding requests and CPU usage as the load metric, respectively, while the rest use other metrics.

To determine a server’s load, a client has two options: 1) Poll the server for its load right before deciding whether to send a request to the server, which incurs additional overhead and latency. 2) Have the server include its load information on its responses and then cache it at the client for later reuse, which is efficient but may result in the client using stale load information and causing load imbalance.

To strike a balance between these two approaches, SR employs an adaptive mechanism. An RPC response is always piggybacked with the server’s current load information. When evaluating a server’s load before sending a new request, the client uses the cached load information only if it is sufficiently fresh (method 1). Otherwise, it polls the server for its realtime load if the network RTT to the server is low compared with the server’s average request-processing time (method 2). In the worst case that the cached load information is stale and the polling overhead is high, it chooses one of the two candidate servers at random. (method 3).

**Design alternative.** LI [13] attempts to solve the load-estimation problem by using methods 1 and 3 alone, without method 2 (polling). Data from our production system show that, with SR’s adaptive mechanism, about 50%, 25%, and 25% of RPC requests end up using methods 1, 2, and 3, respectively. This confirms the usefulness of introducing the just-in-time polling method.

## 4 Evaluation

Our evaluation attempts to answer the following questions:

1. Does SR scale well? (§ 4.1)
2. To what extent does SRLib save hardware costs, and when should one use SRProxy versus SRLib? (§ 4.2)
3. Can SR balance load within and across regions? (§ 4.3)
4. Are sharded services important, and can SR effectively support both sharded and unsharded services? (§ 4.4)

### 4.1 Scalability

Hyperscale is a key design goal that distinguishes SR from most of the existing service meshes. SR currently operates in tens of datacenter regions and runs millions of L7 routers to serve tens of thousands of services. GRS globally distributes service discovery information for millions of containers and hundreds of millions of shards.

To understand the scale of individual services, we plot the number of servers used by services in Figure 8. A small fraction of services are very large while most are very small. Specifically, while 90% of services each use less than 200 servers, 2% of services each use more than 2,000 servers and the largest service uses about 90K servers. Figure 9 shows the RPS of services. Similarly, while most services have a low RPS, some hyperscale services process billions of RPS. These hyperscale services often demand the highest performance and most sophisticated features from SR. Overall, Figures 8 and 9 show that SR scales well for both a small number of hyperscale services and a large number of small services.

In SR’s overall architecture (Figure 6), the central RIB enables separation of concerns for different components in the

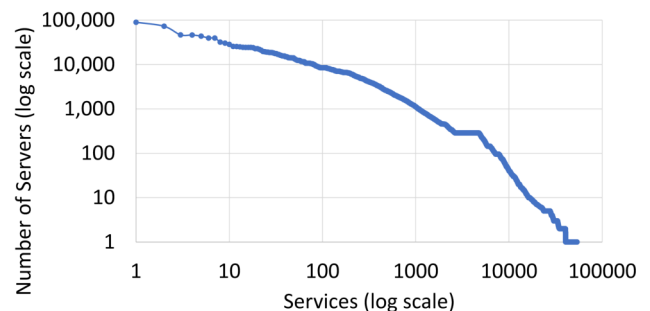


Figure 8: Number of servers used by services. Each dot represents one service. Note that both axes are in log scale.



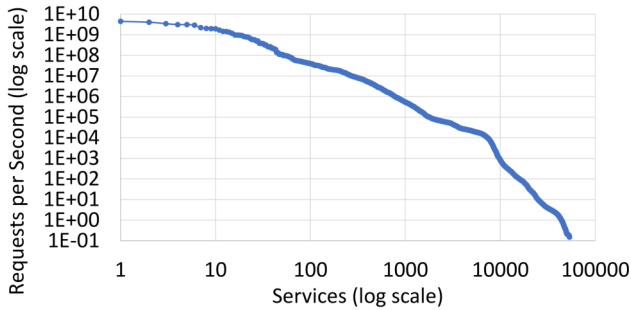


Figure 9: Requests per second by services. Each dot represents one service. Note that both axes are in log scale.

data plane and the control plane so that each component can scale out independently. However, RIB itself might become a bottleneck, primarily due to the large amount of service-discovery data stored in RIB and the associated write rate. Currently, RIB’s total size is about 12GB, processing about 335 writes/second at a total data rate of about 39MB/second. The write rate is low because writes are heavily batched; in particular, sometimes thousands of updates for the service-discovery registry are batched into a single write. In the past, we used ZooKeeper as the data store for RIB, which could not scale beyond a few GB, and hence we sharded RIB. Now we use an in-house data store [6] that scales well and there is no urgency to shard RIB further. Overall, currently RIB is not a bottleneck and it can be further sharded to scale out if needed.

The distribution of RIB is fast, far from reaching any scaling bottleneck. We operate about 2,000 RIB replicas globally, which form a 2-layer data distribution tree among themselves. In our production environment, the distribution latency of an RIB update to reach clients in geo-distributed datacenter regions is 400ms/900ms/1300ms at P50/P95/P99, respectively.

Due to the propagation delay of service discovery information, any system that does service discovery and routing, not just SR, will encounter the problem of stale routing information on some clients. In the face of stale routing information, SR guarantees correctness and strives to minimize performance impact. For example, if an SR client sends a request for a specific shard to a server that no longer holds the shard, the client will receive an error and automatically retry a different server. To improve performance, SR minimizes the chance of this scenario by implementing graceful shard migration. When migrating a shard from server  $X$  to server  $Y$ , as described in our previous work [34], the shard manager first starts the shard on  $Y$ , then updates RIB to redirect clients to send traffic to  $Y$ , and finally stops the shard on  $X$ .

As long as RIB scales well, xRS, CMS, LMS, GRS, and the L7 routers can all scale out horizontally. xRS is sharded by service and can scale out horizontally. Computing the routing table for one service only takes about one second. CMS processes about 10,000 routing-configuration changes per day for about 2,500 services, and 99% of those changes

are driven by automation tools. Overall, the rate of writes to CMS is far from reaching any bottleneck.

To understand the nature of routing-configuration changes, we list the types of the most frequent changes on an average day. A data pair ( $X\%/Y$ ) below means that every day  $X\%$  of the total changes are for a specific type, which are applied to  $Y$  number of services. The top types of changes are 1) *processing timeout* (27%/1700), the server-side RPC processing timeout; 2) *locality ring* (30%/700); 3) *traffic shedding* (11%/3), the percentage of traffic to be shed for a given client ID in an overload situation; and 4) *shadow traffic* (6%/100), the percentage of production traffic to be replicated to a test service. These data demonstrate that it is easy to dynamically reconfigure the routing policies for thousands of services at the central CMS. Moreover, it shows that locality ring is an important feature that is frequently tuned for services to achieve the best cross-region routing performance.

## 4.2 Hardware Cost

We compare the CPU overhead of SRLib and SRProxy, and use case studies to illustrate when to use SRProxy.

### 4.2.1 SRLib versus SRProxy

To quantify the hardware cost, we conduct an experiment to compare three RPC setups: 1) *SRLib*, where a client uses SRLib to route requests to a simple service running on 10 machines; 2) *SRProxy*, where a client sends requests to a remote SRProxy, which forwards requests to the servers; and 3) *Thrift*, where a barebone client hard-codes a most efficient way to randomly select one of the 10 servers and invokes it using the Thrift [51] RPC protocol. SRLib and SRProxy’s internal implementation also use Thrift but add extra logic atop it. Therefore, Thrift represents the lower-bound baseline.

In all three setups, the RPC connections are 100% reused to avoid the connection establishment overhead. All servers used in the experiment are located in the same region to minimize the impact of network latency. We use three RPC payload sizes. The *Production* size uses requests and responses of 5.4KB and 6KB, respectively, which are the average sizes of payloads in production. The *Large* and *Small* sizes use payloads that are  $10x$  or  $\frac{1}{10}x$  of the production payload size, respectively. We report in Figure 10 the end-to-end RPC latency and the total CPU instructions executed across the client, proxy (if used), and server when processing one RPC.

Using production-sized payloads, compared with Thrift, SRLib and SRProxy consume 80% and 273% additional CPU cycles, respectively. The overhead is high because this experiment is set up to measure almost the worst case of SRLib and SRProxy. Since the payload’s data type is a trivial string, serialization and deserialization in Thrift take little time. Moreover, both the RPC client and server do not do any processing. Overall, this setup minimizes all other overhead in order to show the worst-case setup for SRLib and SRProxy. In our production environment, when aggregated across all workloads



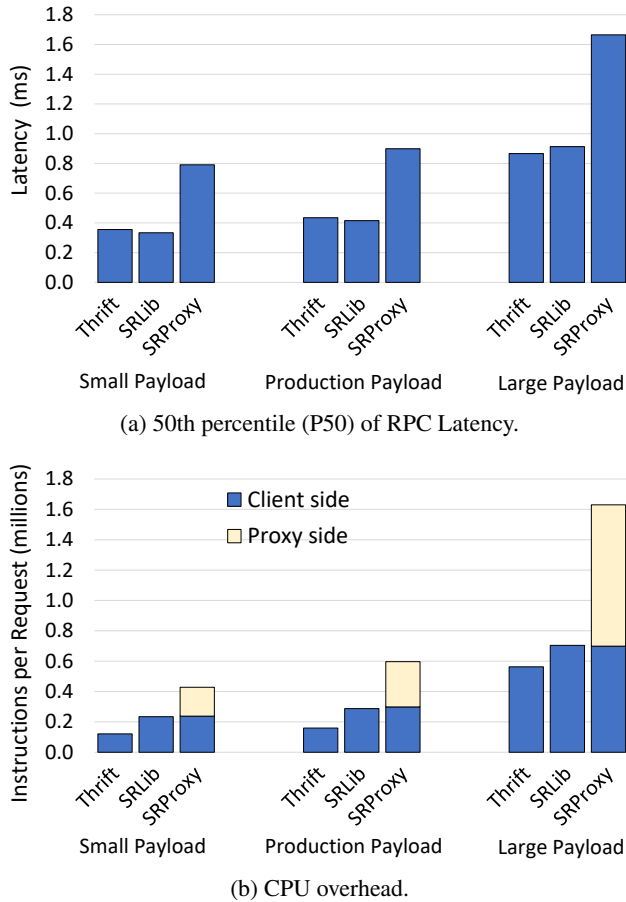


Figure 10: Comparison of latency and CPU overhead across three setups: the Thrift baseline, SRLib, and SRProxy.

running on all servers, SRLib consumes 36% additional CPU cycles compared to Thrift. This is much lower than the 80% overhead observed in this worst-case experiment.

For the SRProxy setup using production-sized payloads, the CPU consumption is evenly split between the client and proxy. Thrift and SRLib have almost identical latency, whereas SRProxy’s latency is 107% higher. Using large-size payloads, the relative overhead of SRLib and SRProxy becomes smaller. Compared with Thrift, SRLib and SRProxy consume additional 25% and 190% CPU cycles, respectively.

This experiment shows that, across the RPC client and proxy, the SRProxy setup in total consumes more than twice the amount of CPU cycles as the SRLib setup. In our production environment, we use thousands of SRProxy machines to route 1.1% of the total RPC requests, which generate only 0.1% of the total RPC data transferred. The remaining RPC requests are routed by SRLib. If we were to completely switch from SRLib to SRProxy and route 100% of the RPC traffic by SRProxy, we would need hundreds of thousands of additional machines for SRProxy.

In the SRProxy setup with production-sized payloads, the split between CPU instructions executed in the kernel and

user space is 26% versus 74%. This indicates that even if the kernel overhead could be entirely eliminated through methods like zero-copy data forwarding, it would still be insufficient to significantly reduce the proxy’s overhead. Moreover, the proxy cannot perform zero-copy data forwarding because it needs to manage encryption and identity.

Using small and production-sized payloads, SRLib’s latency appears to be slightly better than Thrift, but since the standard deviation is high, the small difference is mostly caused by measurement noises in our production network that serves many other production services. Lastly, we would expect to see less CPU cycles consumed by the client side of the SRProxy setup compared with the client side of the SRLib setup, as the former does less routing work. However, the difference is insignificant in this experiment because the SRLib code path is slightly better optimized by our years of investments in it.

#### 4.2.2 Case Study of When to Use SRProxy

As shown in Figure 5, a shared SRProxy improves connection reuse, which potentially can reduce the latency of cross-region RPCs at the expense of extra hardware to host SRProxy. The tradeoff depends on the business value of the reduced latency and the cost of the extra hardware. In practice, we always carefully evaluate our customer’s request of using SRProxy case by case. We present several case studies below.

**E-Comm.** E-Comm is a sharded ranking service used in e-commerce. Due to its tight service-level objective (SLO) for latency, all of its shards were replicated to every region to enable local access. We analyzed its traffic pattern and found that by allowing only 5% of its traffic to go across regions, we could avoid replicating 33% of its shards in every region. This would lead to significant hardware savings but at the expense of increased latency. In Figure 11, we compared E-Comm with and without SRProxy and found that SRProxy improved cross-region connection reuse and reduced the P90 latency from about 325ms to about 150ms. E-Comm’s maximum per-region RPS is about 300K, which can be handled by 4 SRProxy machines since each SRProxy machine can handle about 87K RPS. In practice, about 10 SRProxy machines are needed to provide sufficient buffers for failure and unexpected

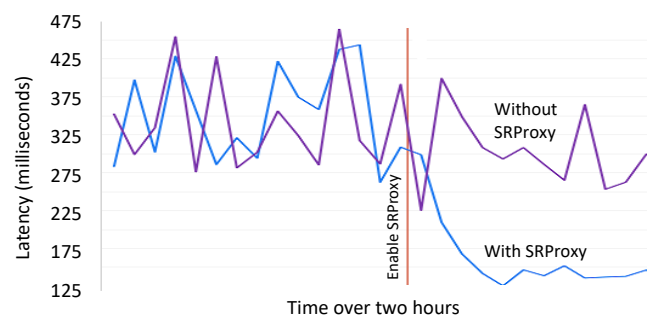


Figure 11: E-Comm’s P90 latency with and without SRProxy.

load spikes. After the evaluation, we decided to enable SRProxy for E-Comm because the 10 SRProxy machines per region would allow us to save 33% of E-Comm’s hardware capacity by routing 5% of its traffic across regions while still keeping its latency within its SLO.

**Key-value store.** This distributed key-value store has 1.5 million data shards. Accordingly, its service-discovery information includes a large shard map for these 1.5 million shards (see an example in Figure 7). It takes a lot of memory on the key-value store’s clients to cache this large service’s full service-discovery information. We evaluated enabling SRProxy to offload the service-discovery cache from the client machines to SRProxy, and noted that it saved on the client machines 250MB memory at P99. Moreover, SRProxy helped with connection reuse and thus latency. The clients have poor connection reuse due to the huge fanout of requests to many different shards hosted by different servers. Shared SRProxies could drastically improve connection reuse and reduce latency by 27% on average. However, due to the key-value store’s high RPS, it would need 1,500 SRProxy machines. Eventually, we decided that the cost would not sufficiently justify the benefits and hence did not use SRProxy to route its traffic.

### 4.3 Load Balancing

SR performs load balancing both within a region and across regions. We evaluate both scenarios in this section.

#### 4.3.1 Same-Region Load Balancing

To evaluate same-region load balancing, we selected 15 representative services that produce significant traffic within a region. 10 of these services are unsharded and 5 are sharded. We measured each service’s average production load (pending requests for unsharded services and CPU usage for sharded services) across its servers and normalized the load by its mean. To evaluate whether the load evenly spreads across

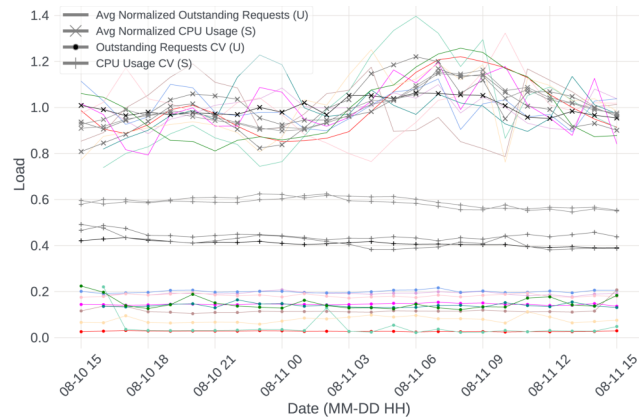


Figure 12: Load balancing within a region for unsharded (U) and sharded (S) services. The top group shows the normalized average load and the bottom two groups show the load’s coefficient of variation (CV) across servers.

a service’s different servers, we calculated the coefficient of variation (CV) for each service. Figure 12 summarizes the results. We observe that the load is concentrated within a narrow band for all services. Across all 15 services, the median CV is low  $P50_{CV} = 0.18$  and  $P95_{CV} = 0.6$ . In particular, the CV for unsharded services is always low ( $P50_{CV}^u = 0.13$  and  $P95_{CV}^u = 0.20$ ), indicating that SR effectively balances the load across their servers.

The CV for sharded services is higher ( $P50_{CV}^s = 0.44$  and  $P95_{CV}^s = 0.61$ ), indicating that the load is less balanced. This is because some shards are hot (receiving a lot of traffic) while others are cold (receiving little traffic), due to the nature of data stored in the shards. As a result, even if SR perfectly routes RPC requests to different replicas of the shards, the load on the servers that host different shards may still be unbalanced. To further balance the load, it may be necessary to migrate shard replicas across containers and/or create additional replicas of the hot shards. However, these operations may have a high overhead, so our shard manager [34] performs these operations only enough to prevent server overload without attempting to perfectly balance the load. Overall, these data show that SR can use a single service mesh to balance load for both sharded and unsharded services.

#### 4.3.2 Cross-Region Load Balancing

**Locality ring.** A service’s locality-ring configuration (§ 3.4.1) guides SR to route requests to nearby servers when appropriate. To assess its effectiveness, we measure P90 latencies for requests that fall into different locality rings. Most services (63.8%) use SR’s default locality-ring configuration: `[same region | neighboring regions | same continent | global]`. Interestingly, 15.4% of services simply set their locality ring as `[global]`, meaning that they have no locality preference. Most of these services are not user facing and not sensitive to latency, but instead care more about availability. 9.7% of services set their locality ring as `[same region | global]`, meaning that they prefer a request being served in the local region, but if that’s impossible, they prefer the request being served by a more lightly loaded server in any region, as opposed to a more heavily loaded server in a nearby region. The remaining 11.1% of services use their own custom locality-ring configuration.

We found that the P90 latency is 12/83/201/262 ms for requests that are served by servers in the *Region / NeighboringRegions / Continent / Global* rings, respectively. This confirms the correct behavior that the inner rings exhibit lower latencies than the outer rings. Moreover, the latency jump at each expanded ring level is significant, indicating that fine-grained locality management is helpful. Initially, our default ring configuration was `[same region | same continent | global]`. As more datacenter regions were added to our infrastructure, the latency difference between regions in the same continent became more significant. Then we were able to easily introduce a new ring level, *neighboring regions*, thanks to the flexibility offered by locality rings.

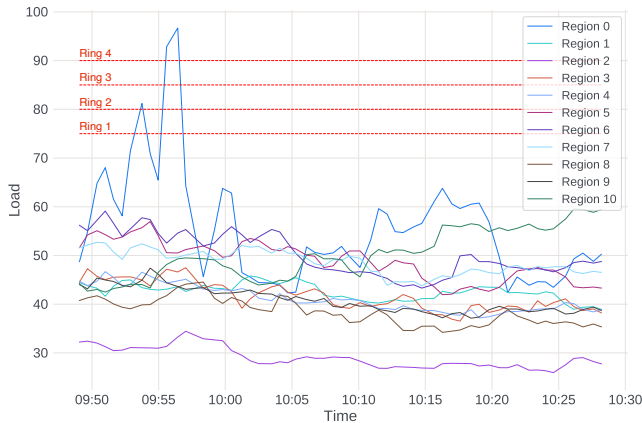


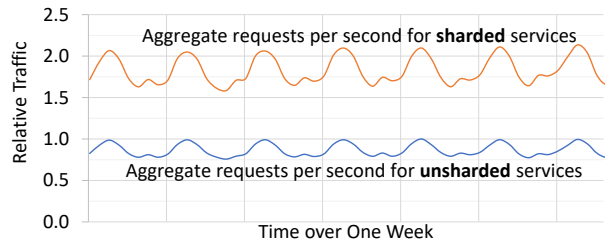
Figure 13: xRS shapes a service’s traffic across several regions to prevent overloads.

**Cross-region load spillover.** xRS computes a cross-region routing table per service to guide L7 routers’ routing decisions (§3.4.1). To evaluate its effectiveness, we choose a service that does newsfeed fetching and ranking for one of our main products. The service uses the following locality-routing thresholds,  $[\text{ring}_1:75\% \mid \text{ring}_2:80\% \mid \text{ring}_3:85\% \mid \text{ring}_4:90\%]$ , where the second number (e.g., 75%) in the pairs is a load threshold. It means that if the measured load in an inner ring exceeds the threshold, xRS should compute a new routing table to shift some traffic from the inner ring to the next-level outer ring in order to reduce load in the inner ring. The load metric for the service is CPU utilization averaged across the service’s all servers in a region.

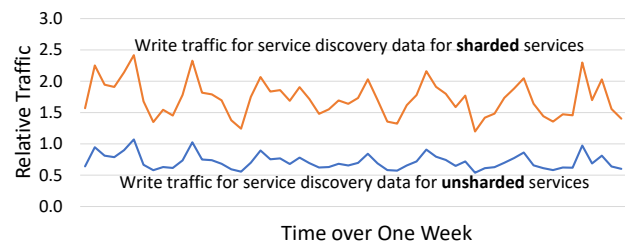
In Figure 13, we report a real incident that happened to the service in production, which was not conducted by us just for the sake of experiment. The figure shows the average load across several regions for the service in the span of 40 minutes. We observe that xRS was able to shift traffic to maintain the load well below the  $\text{ring}_1$  load threshold of 75% for most regions except Region 0 during a short spike.

At 09:53 AM, Region 0 exhibited high load (81.2%), which exceeded its  $\text{ring}_2$  load threshold (80%). xRS evaluated shifting some traffic to Region 0’s  $\text{ring}_2$  regions (i.e., Regions 2, 8, and 10) and chose Region 2 as the target because it had the lowest load. xRS calculated that by shifting some traffic from Region 0 to Region 2, the load of Region 0 would fall below the load threshold. This resulted in a new routing table which reduced  $P_{0,0}$  traffic by 5.35% and set  $P_{0,2} = 5.35\%$ , meaning that 5.35% of requests originating from Region 0 should be routed to Region 2. Then, the load of Region 0 fell to 70.9% and subsequently to 65.47% at 09:54 AM. The corresponding load increase in Region 2 was insignificant because the service had a large capacity footprint in Region 2.

At 09:55 AM, the load of Region 0 spiked again to 92.83% and then to 96.69% at 09:56 AM, which was above the service’s  $\text{ring}_4$  threshold (90%). In response, xRS reduced  $P_{0,0}$



(a) Data plane (i.e., application RPCs).



(b) Control plane (i.e., service-discovery updates).

Figure 14: Traffic for unsharded and sharded services, excluding memcache.

by 12% at first (99.24%  $\rightarrow$  86.92%) and then by another 13% (86.92%  $\rightarrow$  74.38%). The removed traffic was again added to Region 2 alone as it was the least loaded region among all regions in Region 0’s  $\text{ring}_4$ .  $P_{0,2}$  increased first from 0.76% to 13.08% and then from 13.08% to 25.62%. These adjustments helped the load in Region 0 to drop to 64.34% and the region became healthy again at 09:57 AM.

Overall, the whole process above was fully automated by xRS without any manual intervention. It demonstrates that xRS is effective in dynamically managing cross-region traffic.

## 4.4 Sharded Services

Currently, our fleet runs hundreds of sharded services [34]. Although they only account for about 3% of our tens of thousands of services, they generate more traffic than the other 97% unsharded services, because many sharded services are among our largest and highest traffic services. Specifically, most of the largest services in Figures 8 and 9 are sharded, and the two services studied in §4.2.2 are both sharded. To give a sense of scale, our fleet has millions of containers for unsharded services, and hundreds of millions of shard replicas.

Figure 14(a) shows that the aggregate RPS for all sharded services is 212% of the aggregate RPS for all unsharded services. Our memcache [38] is sharded and has the highest RPS among all our services, but it is excluded from the comparison in Figure 14(a) to avoid overshadowing other services. The RPS for memcache alone is 975% of the aggregate RPS for all unsharded services. Although memcache has a high RPS, each of its requests is very lightweight and hence memcache servers do not account for a large fraction of our fleet capacity. Figure 14(b) shows that the aggregate control-plane traffic to

update service-discovery information for all sharded services is 240% of the aggregate traffic for all unsharded services. This is because sharded services more frequently migrate shards across servers to balance load.

Overall, the traffic for sharded services overwhelmingly dominates both the data plane and the control plane of our service mesh, which highlights the importance of providing first-class built-in support for sharded service in a service mesh. Excluding memcache, RPCs for sharded services account for 68% of all RPCs, rising to 92% when memcache is included (as our memcache is sharded). Despite the importance of sharded services, all existing general-purpose service meshes ignore sharded services and exclusively focus on unsharded services. Our key insight in supporting sharded and unsharded services in a single framework is to define a sharding abstraction between SR and services to enforce separation of concerns so that SR can route traffic without knowing a service's internal application logic.

## 5 Limitations of SRLib and Our Solutions

In this section, we discuss several limitations of SRLib and how we address them.

**Dynamic policy updates.** In addition to load balancing, SR offers a large set of service-mesh features such as overload protection [39], observability [1], distributed tracing [32], and encryption. These features are managed through dynamic policy updates, which need to be executed by L7 routers in near-real-time. Without good tooling support, deploying policy updates for a library embedded in applications could be harder than for standalone sidecar proxies. At Meta, this problem is solved by a powerful configuration management system called Configurator [52]. The policies for both SRProxy and SRLib are managed in the same way. When a policy changes, Configurator propagates the change and sends an upcall to SRLib embedded in applications. SRLib then applies the new policy immediately, without restarting the application.

**Source code modification.** One disadvantage of SRLib is that it requires code changes to services. Traditional RPCs use an IP address and a port number to obtain an RPC client, whereas SRLib obtains an RPC client using a service name. The code example below shows that it is straightforward to modify a traditional RPC framework to use SRLib.

```
TraditionalRPCClient *cln = get_client(IP, port);
cln->foo(); // Invoke RPC for foo().
SRClient *cln2 = SR_get_client("service_name");
cln2->foo(); // Invoke RPC for foo().
```

Moreover, source-code modification related to RPC is not unique to SRLib, and is widely adopted by hyperscalers. Regardless of how routing is done, as long as a hyperscaler's RPC framework does not entirely rely on the standard but slow DNS for service discovery, they have to modify their application code to integrate with their custom service-discovery

system. Examples of this include Google's Borg Name Service [55], Netflix's Eureka [17], LinkedIn's Rest.li Dynamic Discovery [49], Twitter's Finagle [19], Uber's Hyperbahn [27], and Airbnb's Synapse [3]. The prevalence of custom service-discovery systems, which often require source-code modifications to use, suggests that this approach is practical as long as the changes are simple and limited to RPC's narrow interface.

**Library code deployment.** Deploying a new version of SRLib is more difficult than deploying a new version of a sidecar proxy. This is because SRLib is compiled into tens of thousands of services, each with its own deployment schedule. Furthermore, in theory, it is possible that some services may not be updated for a long time, resulting in their continued use of an outdated version of SRLib. At Meta, this problem is solved by a powerful continuous software deployment tool called Conveyor [25]. With the help of Conveyor, 97% of the services at Meta are configured to deploy automatically without manual intervention, whether it is on a daily or weekly basis, or whenever a code update successfully passes all tests. Moreover, due to reasons beyond SR, it is a company mandate for all services to be deployed regularly, which ensures that services run with a recent version of SRLib.

**Bugs in SRLib.** If SRLib's new code has a bug, it can be difficult to instantly roll back all services. To mitigate this risk, every major code change or new feature in SRLib is gated by a configuration parameter that can be toggled live in production via Configurator [52], as shown in the example below, without requiring a software deployment or process restart.

```
// Introduce a new FEATURE_X in the SRLib code.
if (check_gate(FEATURE_X)) {
    // New code path...
} else {
    // Old code path...
}
```

In the example above, when `FEATURE_X` is updated on a central server via Configurator, the new parameter value is propagated to all SRLib instances within seconds. SRLib's next invocation to `check_gate(FEATURE_X)` returns the updated parameter value and switches the code path accordingly, without requiring a restart of the application process.

After the above new code is released into production, `check_gate(FEATURE_X)` defaults to false, as if the new code path does not exist. Configurator then manages a canary testing process where it selectively enables the new code path for a small number of replicas of a few services by setting `check_gate(FEATURE_X)` to true. If the test is successful, the new code path is gradually enabled for more services. If a bug is encountered, `FEATURE_X` can be instantly disabled for all services via a configuration change. Overall, incremental rollouts of new SRLib code gated by configuration changes allow us to mitigate the risk of SRLib bugs.



**Summary.** At Meta, managing widely deployed libraries (WDL) such as SRLib is largely a solved problem thanks to the help of Configurator [52] and Conveyor [25]. These tools also help manage about a dozen other WDLs, so the problem is not unique to SRLib. However, we acknowledge that, even with the help of Configurator and Conveyor, it is still more challenging to develop, deploy, and manage SRLib than sidecar or remote proxies because SRLib is linked into every service. Although SR supports both SRProxy and SRLib, we prioritize the cost savings of hundreds of thousands of machines that come with the routing-library approach, over the simplicity that comes with the proxy approach. Our experience in production demonstrates that the routing-library approach is not only cost-effective but also practical, even in highly complex environments, despite its challenges.

## 6 Related Work

There is an array of works from both academia and industry discussing routing and load balancing in datacenter environments, at either layer-3/4 [5, 8, 12, 14, 18, 21, 40, 44, 46] or layer 7 [3, 5, 15, 19, 20, 23, 26, 30, 36, 37, 43, 48]. Layer-3/4 load balancers can be implemented either in hardware [8, 21, 40] or in software [5, 14, 22, 28, 44, 46, 48]. As a layer-3 solution, anycast [56] can route requests to nearby servers, but it does not consider the servers' dynamic load. As shown in Figure 14, the majority of our traffic is for sharded services, which cannot be handled by these layer-3/4 solutions as they do not understand application shards.

More relevant to SR are layer-7 (L7) service-mesh solutions that route requests across microservices. L7 routing can inspect application-level information, enabling more advanced load balancing. L7 routing can be performed by a group of dedicated proxies [3, 5, 15, 20]. However, using remote proxies comes with significant latency and hardware costs, so SR limits the use of SRProxy to around 1% of its traffic, only for services that can benefit the most from connection reuse.

More related to SRLib, which routes 99% of our traffic, are service meshes that distribute L7 decisions closer to the clients. eBPF [35] is efficient but is limited in its L7 capabilities. For example, Cilium [29]'s eBPF program can only handle L3/L4 protocols, and it still needs to use a sidecar proxy to handle L7 protocols. RPC frameworks such as Thrift [51], gRPC [23], and Finagle [19] are the foundations of service meshes, but they do not offer the complete capabilities needed for a geo-distributed service mesh, such as global-traffic-aware routing.

To address these limitations, more complex service meshes [15, 30, 36, 37] have been proposed. Envoy [15] is typically deployed as a sidecar proxy, and Istio [30] provides a control plane to manage Envoy proxies. We compare different service meshes in Table 2 and show that the sidecar approach is easy to deploy, but increases latency and incurs significant hardware costs. Zhu et al. [57] show that Istio adds 92% extra CPU usage and increases the latency by 185% [57]. mRPC [7] confirms that a sidecar increases the P99 RPC la-

tency by 180% and decreases throughput by 44%. SR takes the routing-library approach to avoid the overhead of a proxy.

mRPC [7] eliminates the double marshaling overhead of the sidecar approach, by using shared memory to communicate between the application and the sidecar and by not performing marshaling in the application. However, this approach requires modifying applications to allocate memory for RPC arguments from a heap in shared memory. This can be difficult since memory allocations tend to scatter throughout an application and sometimes occur in system libraries such as `strdup()` that cannot be easily modified.

While Istio offers locality-aware routing based on static rules [31], SR dynamically computes a per-service global routing table based on global traffic. Google Slicer [2] supports service discovery for sharded services, but this function is not offered by the underlying service mesh out of the box.

## 7 Conclusion

We presented Meta's global service mesh, called *ServiceRouter* (SR). SR differs from other publicly known service meshes in several significant ways. First, SR scales significantly beyond previously published work, currently processing tens of billions of requests per second. This is achieved by massively replicating the routing information base (RIB) to guide L7 routers to self-configure and self-manage in a decentralized manner. Second, SR minimizes hardware costs by providing the service-mesh function out of an embedded routing library for 99% of its traffic, in contrast to the common approach of using sidecar or remote proxies alone. Third, SR introduces the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed datacenter regions. Finally, SR supports both sharded and replicated services through a common underlying routing framework.

Our ongoing work is focused on improving global routing in accordance with global capacity management [16] and enhancing overload protection to ensure services gracefully degrade in the event of large-scale disasters [39].

## Acknowledgments

This paper presents 11 years of work by past and current members of several teams at Meta, including ServiceRouter, CSLB, SMC, and Falcon. In particular, we would like to call out the current members of the ServiceRouter team who are not on the author list: Akrama Baig Mirza, Bo Huang, Emanuele Altieri, Kenny Lau, Lijie Tang, Mikhail Shatalov, Nan Su, Nitesh Kant, Scott Diao, Tao Chen, Wei Song, and Weilun Wang. We thank all reviewers for their insightful comments, Shie Erlich for the support, as well as Andrii Grynenko, Rahul Gokul, and Stepan Palamarchuk for their contributions to some ideas presented in the paper.

## References

- [1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.
- [2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 739–753, 2016.
- [3] Airbnb Synapse. <https://github.com/airbnb/synapse>.
- [4] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing: Challenges and algorithms. In *2012 second symposium on network cloud computing and applications*, pages 137–142. IEEE, 2012.
- [5] AWS Elastic Load Balancing. <https://aws.amazon.com/elasticloadbalancing/>.
- [6] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.
- [7] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, Boston, MA, April 2023. USENIX Association.
- [8] Reuven Cohen, Matty Kadosh, Alan Lo, and Qasem Sayah. LB Scalability: Achieving the Right Balance Between Being Stateful and Stateless. *IEEE/ACM Transactions on Networking*, 30(1):382–393, 2021.
- [9] Ben Coleman. Rendezvous Hashing Explained, 2020. <https://randorithms.com/2020/12/26/rendezvous-hashing.html>.
- [10] Consul. <https://www.consul.io>.
- [11] Robert B Cooper. Queueing theory. In *Proceedings of the ACM’81 conference*, pages 119–122, 1981.
- [12] Alejandro Forero Cuervo. Load Balancing in the Datacenter, 2016. <https://sre.google/sre-book/load-balancing-datacenter/>.
- [13] Michael Dahlin. Interpreting stale load information. *IEEE Transactions on parallel and distributed systems*, 11(10):1033–1047, 2000.
- [14] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.
- [15] Envoy Proxy. <https://www.envoyproxy.io/>.
- [16] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.
- [17] Eureka. <https://github.com/Netflix/eureka>.
- [18] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google’s Software-Defined Networking Control Plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98, 2021.
- [19] Finagle: A Protocol-Agnostic RPC System. [https://blog.twitter.com/engineering/en\\_us/a/2011/finagle-a-protocol-agnostic-rpc-system](https://blog.twitter.com/engineering/en_us/a/2011/finagle-a-protocol-agnostic-rpc-system).
- [20] Rohan Gandhi, Y Charlie Hu, and Ming Zhang. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
- [21] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.
- [22] Introducing the GitHub Load Balancer. <https://githubengineering.com/introducing-glb/>.
- [23] gRPC. <https://grpc.io/>.

- [24] gRPC Lookaside Load Balancer. <https://github.com/markitdigital/grpc-lookaside>.
- [25] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.
- [26] HAProxy. <https://www.haproxy.com/>.
- [27] Hyperbahn. <https://github.com/uber-archive/hyperbahn>.
- [28] IPVS Software - Advanced Layer-4 Switching. <http://www.linuxvirtualserver.org/software/ipvs.html>.
- [29] Isovalent. Cilium Service Mesh—Everything You Need to Know, 2022. <https://isovalent.com/blog/post/cilium-service-mesh/>.
- [30] Istio. <https://istio.io/>.
- [31] Istio Locality Load Balancing. <https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/>.
- [32] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O’Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.
- [33] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.
- [34] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.
- [35] Idit Levine, Christian Posta, and Yuval Kohavi. eBPF for Service Mesh? Yes, but Envoy Proxy is here to stay, 2021. <https://www.solo.io/blog/ebpf-for-service-mesh/>.
- [36] Chien-Chih Liao, Pawel Krolikowski, and Sangeeta Kundu. Better Load Balancing: Real-Time Dynamic Subsetting, 2022. <https://www.uber.com/blog/better-load-balancing-real-time-dynamic-subsetting/>.
- [37] Linkerd. <https://linkerd.io/>.
- [38] Memache. <https://memcached.org/>.
- [39] Justin Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Nazim Uddin, Chad Nachappan, Sari Tran, Shuyang Shi, Tina Luo, Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. Defcon: Preventing Overload with Graceful Feature Degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.
- [40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.
- [41] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [42] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O’Reilly Media, Inc.", 2016.
- [43] Netflix Ribbon. <https://github.com/Netflix/ribbon>.
- [44] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.
- [45] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Alaa Youssef. Managing the response time for multi-tiered web applications. *IBM TJ Watson Research Center, Yorktown, NY, Tech. Rep. RC23651*, 2005.
- [46] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM ’13, pages 207–218, New York, NY, USA, 2013. ACM.

- [47] Performance and Scalability of Istio. <https://istio.io/latest/docs/ops/deployment/performance-and-scalability/>.
- [48] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.
- [49] Rest.li Dynamic Discovery. [https://linkedin.github.io/rest.li/Dynamic\\_Discovery](https://linkedin.github.io/rest.li/Dynamic_Discovery).
- [50] Daniel E Rivera, Manfred Morari, and Sigurd Skogestad. Internal model control: PID controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265, 1986.
- [51] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.
- [52] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatchalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.
- [53] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutorenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 787–803. USENIX Association, 2020.
- [54] David Thaler and China V Ravishankar. A name-based mapping scheme for rendezvous. In *Technical Report CSE-TR-316-96, University of Michigan*, 1996.
- [55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.
- [56] Scott Weber and Liang Cheng. A survey of anycast in ipv6 networks. *IEEE Communications Magazine*, 42(1):127–132, 2004.
- [57] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. In *arXiv preprint arXiv:2207.00592*, 2022. <https://arxiv.org/pdf/2207.00592.pdf>.





# Characterizing Off-path SmartNIC for Accelerating Distributed Systems

Xingda Wei<sup>1,2</sup>, Rongxin Cheng<sup>1,2</sup>, Yuhan Yang<sup>1</sup>, Rong Chen<sup>1,2</sup>, and Haibo Chen<sup>1</sup>

<sup>1</sup>Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

<sup>2</sup>Shanghai AI Laboratory

## Abstract

SmartNICs have recently emerged as an appealing device for accelerating distributed systems. However, there has not been a comprehensive characterization of SmartNICs, and existing designs typically only leverage a single communication path for workload offloading. This paper presents the first holistic study of a representative off-path SmartNIC, specifically the Bluefield-2, from a communication-path perspective. Our experimental study systematically explores the key performance characteristics of communication among the client, on-board SoC, and host, and offers insightful findings and advice for designers. Moreover, we propose the concurrent use of multiple communication paths of a SmartNIC and present a pioneering guideline to expose new optimization opportunities for various distributed systems. To demonstrate the effectiveness of our approach, we conducted case studies on a SmartNIC-based distributed file system (LineFS) and an RDMA-based disaggregated key-value store (DrTM-KV). Our experimental results show improvements of up to 30% and 25% for LineFS and DrTM-KV, respectively.

## 1 Introduction

Remote Direct Memory Access (RDMA) has been widely adopted in modern data centers [23, 71, 20], pushing network bandwidth (towards 400 Gbps [44]) and distributed system performance [17, 76, 75, 64, 80, 82, 77] to the next level. However, the high-speed network requires more CPU resources to saturate a fast RDMA-capable NIC (RNIC) [38], which places a significant CPU burden on distributed systems [32]. One-sided RDMA can alleviate CPU pressures by enabling the RNIC to directly read and write host memory in a CPU-bypass way. However, the limited offloading capabilities may cause network amplifications and thus degrade system performance [61, 28].

The continuous improvements in RDMA [67] and the essential power and memory walls of CPUs have led to the emergence of SmartNICs—the RNICs with programmable capabilities. These NICs offer systems the opportunity to offload more complex computations to the NIC. Currently, there are two main types of SmartNICs. The first one is the *on-path* SmartNIC [42], which directly exposes the processing units (NIC cores) for handling RDMA packets to the systems. Unfortunately, programming low-level NIC cores with firmware [38, 61] and isolating the offloaded program

from normal RDMA requests pose significant burdens on developers. To simplify system development, the *off-path* SmartNIC [52, 53, 9, 51] attaches a programmable multicore SoC (with DRAM) next to the RNIC cores, which is off the critical path of RDMA. Thanks to this separation, the SoC is independent of normal RDMA requests and can further deploy a full-fledged OS to make the developments easier [32]. Specifically, developers can treat the SoC as a separate server. In this paper, we focus on off-path SmartNIC<sup>1</sup> due to its generality and programmability.

There have been valuable studies on characterizing off-path SmartNICs [38, 37, 32, 68, 2], with a focus on their ability to offload computation. A key finding is that the computing power of off-path SmartNICs is weaker than that of the host [38, 37, 32]. This means that off-path SmartNICs do not improve the speed of a single network path, such as that between NIC and the host. For example, iPipe [38] found that the path between the host and SoC has a relatively high latency due to the support for more developer-friendly RDMA.

Although prior work has been valuable in utilizing SmartNICs for distributed systems, it has primarily focused on offloading computation to the SmartNIC’s SoC. However, it is surprising that the fundamental function of SmartNICs, namely networking, has been overlooked despite its significant impact on overall performance. In fact, networking on the SmartNIC is intricate, because it provides multiple communication paths. For example, SmartNICs support using RDMA to access the memory of the host or SoC, as well as exchanging data between the host and the SoC.

To this end, this paper conducts the first systematic study on characterizing the performance of communication paths of SmartNIC. Unlike previous studies that simply report basic performance numbers [37, 32, 68], we systematically analyze the performance implications of SmartNIC architecture on different paths. Specifically, we investigate why and when one path may be faster than another, identify the bottlenecks for each path, examine how the heterogeneity of the SoC brings performance anomalies in paths related to the SoC, and finally explore how paths interact with each other. The main highlights of our results are:

- *Different paths exhibit diverse performance characteristics.* The RDMA path from the NIC to the SoC is up to 1.48× faster than the path to the host.

<sup>1</sup>This paper will use “SmartNIC” (or “SNIC” for brevity) to specifically refer to off-path SmartNICs.

- *The SoC introduces new performance anomalies to paths related to it.* The low-level hardware details of the SoC, including the memory access path and PCIe MTU, differ from those of the more powerful host CPU. Without considering such factors, RDMA requests involving the SoC suffer from up to 48% bandwidth degradation.
- *The paths between the SoC and the host may underutilize the PCIe.* RDMA from the SoC to the host (and vice versa) crosses the NIC internal PCIe twice. It can only utilize half of the PCIe bandwidth and requires processing up to 6× more PCIe packets than the others. DMA only passes the PCIe once, but it is not always faster than RDMA due to the weaker SoC DMA engine (compared to the one on the RNIC) and also suffers from packet amplifications.

Based on our performance characterization, we found that prior approaches, which mainly optimize a single path for a specific functionality of distributed systems, failed to fully exploit SmartNICs. This is because a single path cannot utilize the computing and networking capability of SmartNICs. Further, only considering a single path may ignore resource interference between different paths (e.g., the PCIe and PCIe switches). As a result, LineFS can only utilize up to 117 Gbps of bandwidth on a 200 Gbps SmartNIC. A similar issue exists in SmartNIC-based disaggregated key-value store: while choosing a path to offload all key-value (KV) store operations to the SmartNIC SoC can eliminate the network amplification in existing RDMA-based key-value stores, the wimpier computing power of SmartNIC SoC limits its overall throughput.

Based on the observations from our study, we further propose an optimization guideline to help designers smartly exploiting multiple paths of SmartNICs. Instead of optimizing distributed systems along a single path, it holistically exploits multiple paths for functionalities with different characteristics and carefully considers cross-path interference. To demonstrate the efficacy of our guideline, we conduct two case studies by optimizing two state-of-the-art systems, namely LineFS [32] and DrTM-KV [11, 76]. Due to the exposed new optimization spaces, following our guideline can improve the performance of LineFS and DrTM-KV by up to 30% and 25% accordingly.

**Contributions.** We summarize our contributions as follows:

- A comprehensive performance characterization of representative off-path SmartNICs, with a particular focus on various communication paths.
- The first optimization guideline for smartly exploiting the multiple paths of SmartNICs with managed cross-path resource interference.
- Two case studies on SmartNIC-accelerated distributed systems (i.e., file system and key-value store) with notable performance improvements, demonstrating the efficacy of our guideline.

**Assumptions and generalizability of our work.** We assume an off-path SmartNIC with the following architecture: the SoC is linked with NIC cores via a PCIe switch, and there is heterogeneity between SoC and host CPUs. We believe this is a representative architecture, as many older (e.g., NVIDIA Bluefield-1 [55], Broadcom Stingray [9]), current (e.g., NVIDIA InnoVA2 [51], Bluefield-2 [52]), and upcoming SmartNICs (e.g., Bluefield-3 [53], Marvell OCTEON 10 DPU [43]) use a similar setup. We conducted experiments on Bluefield-2 [52]—the state-of-the-art SmartNIC with this architecture. Meanwhile, we also confirmed that our results hold for Bluefield-1.

However, we acknowledge that significant architectural changes (e.g., on-path SmartNICs) may affect our findings. Nevertheless, we argue that our methodology—first studying the performance implications of each communication path and then smartly exploiting multiple paths of SmartNICs—can be generalized to other SmartNICs. Our benchmarking code, tools, and systems are available at <https://github.com/smarnickit-project>.

## 2 Background and Context

### 2.1 RDMA-capable NICs (RNICs)

RDMA (Remote Direct Memory Access) is a low-latency (2  $\mu$ s) and high-bandwidth (200 Gbps) network widely adopted in modern data centers [23]. One intuitive way to utilize RDMA is to accelerate message passing with its *two-sided* primitives (SEND/REC<sup>2</sup>), such as RDMA-based RPC [27, 17, 12, 47, 30]. Alternatively, the one-sided primitives (READ/WRITE<sup>2</sup>) allow the RNIC to access the host memory bypassing the host CPU. Specifically, the NIC core internally uses the direct memory access (DMA) feature of the PCIe link to access the host memory (see Figure 1(a)).

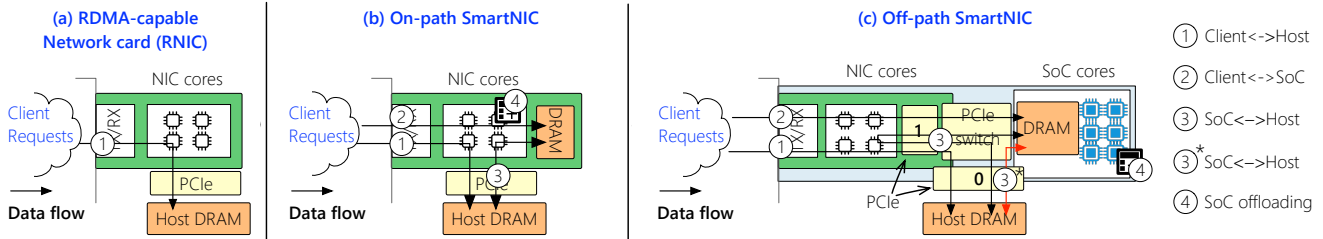
Though RDMA has boosted the performance of many distributed systems [18, 76, 62, 29], usually by orders of magnitude, it still has the following two problems especially when the RNICs scale up to higher performance.

**Issue #1: Host CPU occupation.** For two-sided primitives, distributed systems need non-trivial CPUs to saturate a powerful NIC. Our measurements show that a 24-core server can only saturate 87 million packets per second (Mpps) on a 200 Gbps RNIC (ConnectX-6), while NIC cores can process more than 195 Mpps.<sup>3</sup> A recent work further shows that a distributed file system requires 2.27× CPU cores to handle network packets, when the network bandwidth scales from 25 Gbps to 100 Gbps [32]. Although deploying more powerful CPUs can alleviate this issue, RNIC bandwidth is also rapidly growing, currently reaching up to 400 Gbps [44].

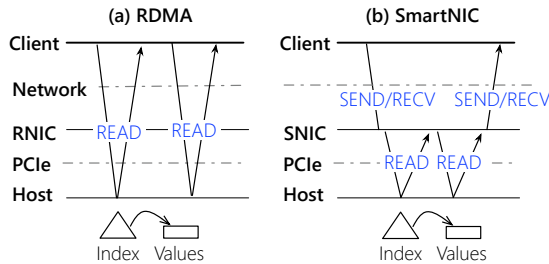
**Issue #2: Network amplification.** Using one-sided RDMA primitives alleviates the host CPU pressure by allowing sys-

<sup>2</sup>We use READ/WRITE to indicate RDMA READ/WRITE in this paper.

<sup>3</sup>Detailed hardware setups can be found in §2.4.



**Figure 1:** Architecture of different NICs: (a) RDMA-capable NIC (RNIC), (b) on-path SmartNIC, and (c) off-path SmartNIC (our focus).



**Figure 2:** An illustration of a get request in a distributed in-memory key-value store that is accelerated by using either (a) RNICs (w/ network amplification) or (b) SNICs (w/o network amplification).

tems to offload memory accesses to the RNIC. However, the limited offloading capability constraints system performance, as a single request may involve multiple round trips of READs/WRITEs to complete (usually termed *network amplification*). Figure 2(a) exemplifies the execution of a *get* request on a distributed in-memory key-value store with one-sided RDMA READs. The client first uses one (or multiple) READ(s) to query the index for a given key. Based on the index returned by the previous READs, an additional READ is issued to retrieve the value.

## 2.2 From RNICs to SmartNICs

To address the limitations of RNICs, SmartNIC adds an on-board memory (4–64 GB) together with various computation units (e.g., SoC) to the NIC. By exposing them to the developers, SmartNIC enables offloading customized computations onto it. Specifically, SmartNICs can be categorized as follows.

**On-path SmartNIC.** As shown in Figure 1(b), the on-path SmartNIC exposes the NIC cores to the systems with low-level programmable interfaces, allowing them to directly manipulate the raw packets. As the name implies, the offloaded code is *on* the critical path of the network processing pipeline. Example NICs include Marvell LiquidIO [42] and Netronome Agilio [48]. The benefit is that the offloaded code is closer to the network packets. Therefore, inline requests that only interact with the NIC, such as writing to the on-board memory (②), are extremely efficient [38, 61].

However, on-path SmartNIC has two limitations. First, the offloaded code (④) competes NIC cores with the network requests sent to the host (①). If offloading too much computation onto it, the normal networking requests sent to the host

**Table 1:** Hardware description of Bluefield-2 [52].

Component	Hardware description
NIC cores	ConnectX-6 (2× 100 Gbps RDMA ports)
SoC cores	ARM Cortex-A72 processor (8 cores, 2.75 GHz)
SoC memory	1× 16 GB of DDR4-1600 DRAM
PCIe1	PCIe 4.0 × 16 (256 Gbps bandwidth)

would suffer a significant degradation [38]. Second, programming on-path NICs is difficult due to its low-level interface.

**Off-path SmartNIC.** As shown in Figure 1(c), the off-path SmartNIC offers an alternative: it packages additional compute cores and memory in a separate SoC next to the NIC cores. Therefore, the offloaded code is *off* the critical path of the network processing pipeline. From the NIC perspective, the SoC can be viewed as a second full-fledged host with an exclusive network interface. To bridge the NIC cores, SoC and host together, a PCIe switch is integrated inside the SmartNIC to properly dispatch network packets. Example NICs include NVIDIA Bluefield [52, 53] and Broadcom Stingray [9].

Compared to the on-path counterparts, the offloaded code does not affect the network performance of the host as long as it does not involve network communications (②). Thanks to this clear separation, the SoC can run a full-fledged kernel (e.g., Linux) with a full network stack (i.e., RDMA), simplifying system development and allowing for offloading complex tasks [32]. However, accelerating distributed systems with off-path SmartNICs is typically more challenging than using the on-path counterparts. This is because the PCIe switch prolongs all communication paths (i.e., ①, ②, and ③), causing potential performance degradation.

## 2.3 Target SmartNIC: NVIDIA Bluefield-2

We conduct our study on Bluefield-2, a typical off-path SmartNIC optimized for offloading general-purpose computations. Figure 1(c) illustrates its overall hardware architecture, with detailed hardware configuration shown in Table 1.

**Hardware.** Bluefield-2 equips a mature RNIC (ConnectX-6) as its NIC cores for high-speed networking. These cores support all RDMA operations. Its programmability comes from an integrated on-board SoC, which has 16 GB DRAM and an ARM Cortex-A72 (8 cores, 2.75 GHz). A PCIe 4.0 switch bridges the NIC cores, SoC and host together, enabling



**Table 2:** Machine configurations in our two rack-scale RDMA-capable clusters.

Name	Nodes	RDMA-capable NIC	Host PCIe (PCIe0)	Host CPU	Host Memory
SRV	3	1 × ConnectX-6 (200 Gbps) 1 × Bluefield-2 (200 Gbps)	PCIe 4.0 × 16 (256 Gbps)	2 × Gold 5317 v4 (12 cores, 3.6 GHz)	128 GB DDR4-2933
CLI	20	1 × ConnectX-4 (100 Gbps)	PCIe 3.0 × 16 (128 Gbps)	2 × E5-2650 v4 (12 cores, 2.2 GHz)	96 GB DDR4-1600

bi-direction data transfer of up to 256 Gbps. Note that the SoC is linked to the PCIe switch via an internal link, rather than through PCIe.<sup>4</sup> Specifically, the hardware counters provided by Bluefield [54] also imply that it has only two PCIe links: one linking RNIC with the switch (PCIe1) and the other linking the switch with the host (PCIe0).

**Software.** The SoC runs a full-fledged Linux, allowing developers to treat it as a normal ARM server. The kernel also hosts a full RDMA stack, making it convenient for enabling RDMA-based communication. In addition, Bluefield provides DOCA [57] SDK for advanced usage, such as DMA.

**Communication primitives: RDMA and DMA.** All communication paths related to the SoC are conducted using RDMA to simplify system development. As shown in Figure 1(c), clients can issue one-sided or two-sided RDMA requests to the SoC (②), similar to a twin server on the host. Meanwhile, the SoC can also interact with the host via RDMA, and vice versa (③). However, exchanging data between the SoC and the host must pass through the RNIC (PCIe1 and NIC cores) for RDMA support, which adds a hidden bottleneck to this path. Fortunately, we found that Bluefield further provides DMA support (③\*) with DOCA [57], allowing the SoC to use DMA to access the host memory (and vice versa), bypassing the RNIC.

**Existing state of exploring Bluefield.** Previous studies [38, 37, 32, 68] have mainly focused on the computing power of Bluefield (④ in Figure 1), revealing the relative weakness of the SoC cores in terms of performing offloaded tasks and sending network requests. This is because the frequency and number of cores are inferior to those of the host CPU. Due to the power constraints of SmartNICs, it is unlikely that the relative performance comparison between the NIC and host CPU will change. Hence, we take this as a premise during our investigation.

In contrast, few studies have considered various communication patterns in Bluefield (i.e., ①, ②, and ③), which are the main focus of our work. Thostrup *et al.* [68] found that accessing the SoC memory (②) using READ is faster than accessing the host memory (①) in the same way. iPipe [38] shows that using RDMA to communicate between the host and SoC (③) has high latency due to the software overhead of supporting RDMA. This paper systematically explores the performance characteristics of Bluefield and summarizes insightful lessons and advice for future system developers.

<sup>4</sup>This has been confirmed by the NIC vendor.

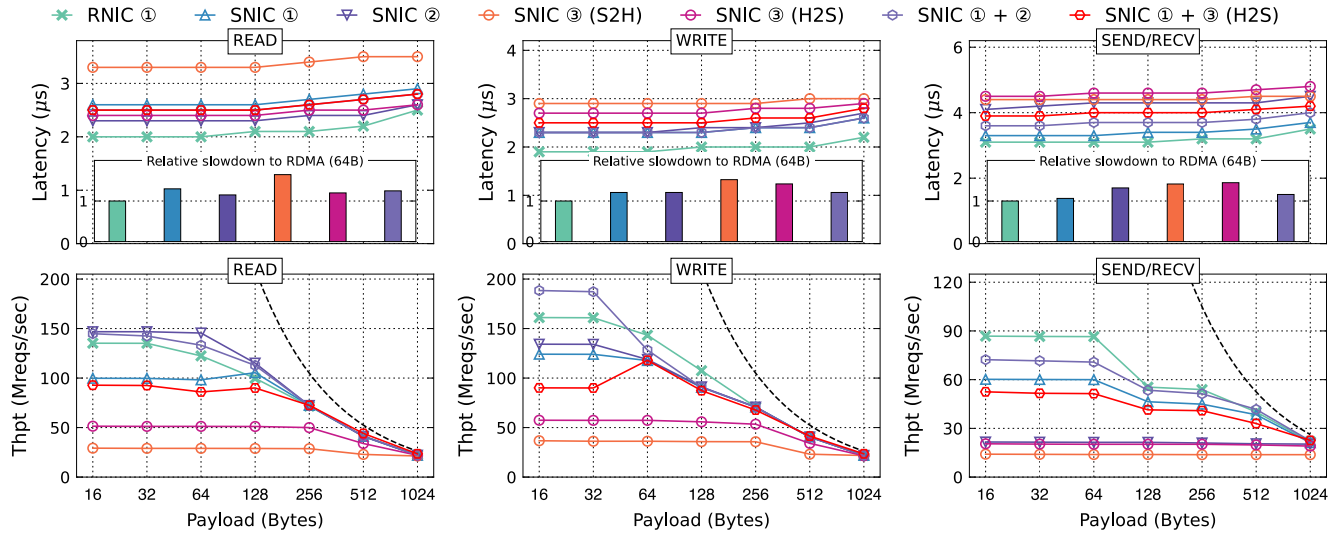
## 2.4 Notation and testbed

**Notations.** This paper follows Bluefield’s hardware specification when describing low-level hardware details related to Bluefield-2. As shown in Figure 1(c), “PCIe1” refers to the PCIe link connecting the NIC cores to the PCIe switch, and “PCIe0” refers to the link connecting the switch and the host’s PCIe controller. The ARM cores, along with the on-chip memory of Bluefield-2, are collectively referred to as “SoC.” The machine hosting Bluefield-2 is referred as the “host.” Furthermore, we use the terms “requester” and “responder” to refer to the machine issuing the RDMA requests and the destination hardware component, respectively. For example, in Figure 1(c), the requesters of paths ① and ② are any RDMA-capable machines (also called clients), and the responders are the host and SoC, respectively. For path ③, the requester and responder are the host and SoC, respectively, and vice versa.

**Testbed.** Table 2 presents the machine configurations in our testbed. To best utilize SmartNIC, we deploy Bluefield-2 on the servers (SRV) with matching PCIe link (PCIe 4.0) by default. These machines can replace Bluefield-2 with 200 Gbps ConnectX-6 (RNIC) for comparisons. Other machines (CLIs) serve as clients that issue RDMA requests to the servers. All machines in SRV and CLIs are connected through a Mellanox SB7890 100 Gbps InfiniBand Switch. Note that the network performance of the evaluated 200 Gbps NIC is not limited since they connect to the switch with two 100 Gbps ports.

**Table 3:** The findings and advice from our study. Claims supported by sufficient evidence are denoted by **E**, while those supported by hypotheses are denoted by **H**.

SNIC Paths	Findings/Advice	E/H
① (§3.1)	Throughput of RDMA is lower than RNIC	H
	Latency of RDMA is higher than RNIC	E
② (§3.2)	One-sided RDMA performance is better	H
	Avoid memory accesses to close addresses	E
	Avoid large READ requests	H
③/③* (§3.3)	RDMA overuses the PCIe bandwidth	E
	Avoid large READ/WRTIE requests	H
	Enable doorbell batching carefully for RDMA	E
	Use DMA (③*) to improve PCIe utilization	E
①+② (§4.1)	Improve throughput by using paths ① and ② concurrently (esp. in opposite directions)	H
①/②+③ (§4.1)	Selectively offload traffic to ③	E



**Figure 3:** The end-to-end latency (upper) and peak throughput (lower) of random inbound RDMA requests on different NICs. The symbols ①, ②, and ③ in the legend correspond to the communication paths listed in Figure 1.

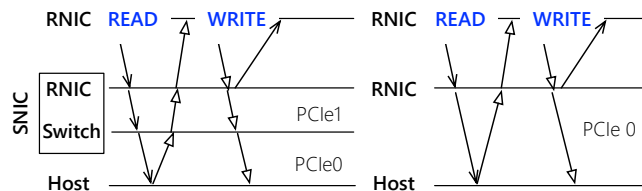
### 3 Characterizing SmartNIC Performance

As mentioned in §2.3, it is well-known that the *computing power* of NIC is wimpier than that of the host CPU. Therefore, we focus on analyzing the *communication efficiency* of SmartNIC. Figure 3 shows the end-to-end latency and peak throughput of sending different RDMA requests (e.g., READ, WRITE, and SEND/RECV) using either RNIC or SmartNIC through different communication paths.

**Evaluation setup.** We conducted our experiments on the clusters described in Table 2, using a state-of-the-art RDMA communication framework [76]. For one-sided operations (READ and WRITE), the requester communicates with one responder using RDMA’s reliable connection (RC) queue pairs (QPs). The responder addresses are randomly chosen from a 10 GB address space by default. For two-sided operations (SEND/RECV), the responder implements an echo server that utilizes all available cores for handling messages, and the requester communicates with it via unreliable datagram (UD) QPs for better performance [29, 76, 30]. For end-to-end latency, we deploy one requester machine to prevent interferences from queuing effects. For peak throughput, we use up to eleven requester machines to saturate the responder. Finally, we enable all well-known optimizations, including address alignment [81], unsignaled requests [27] and huge pages [17] to prevent side effects from misusing RDMA.

#### 3.1 Communication from Client to Host (path ①)

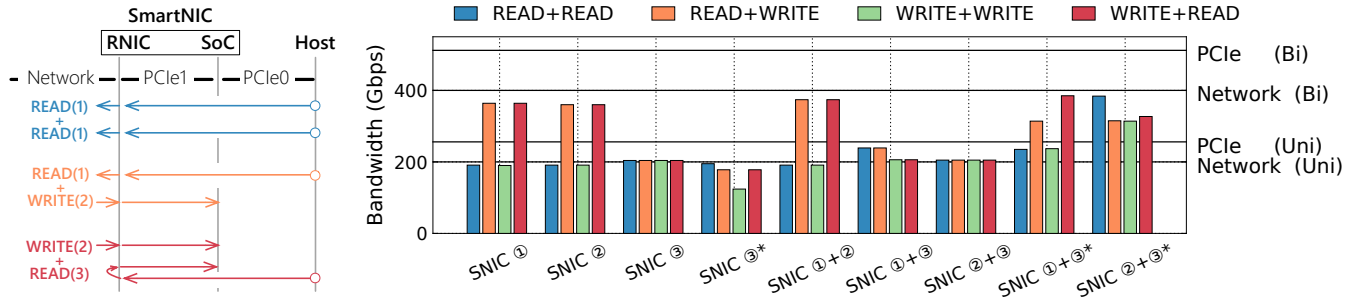
**Latency.** To compare communication with the host, we conduct an apple-to-apple comparison between Bluefield-2 (SNIC ①) with ConnectX-6 (RNIC ①), as they share the same NIC cores [52]. Their performance gap best illustrates the “performance tax” paid by the SmartNIC architecture. As shown in Figure 3, SNIC ① has 15–30%, 15–21%, and



**Figure 4:** The exec. flow of READ/WRITE on SNIC and RNIC.

6–9% higher latency than RNIC ① for READ, WRITE, and SEND/RECV, respectively. The increased latency on SNIC comes mainly from the PCIe switch and PCIe1 between the host and NIC cores. The one-way PCIe latency is approximately 300 ns, which is non-trivial for small RDMA requests (1–2 μs). Note that the result is measured indirectly. Specifically, the end-to-end read latency on SNIC and RNIC is 2.6 μs and 2.0 μs, respectively. Compared to RNIC, READ on SNIC passes through the PCIe switch twice (see Figure 4). Thus, the cost of each pass is around 300 ns, which matches the number reported in recent literature [69]. Furthermore, the increased latency of WRITE on SNIC is lower than that of READ, because it omits one pass through PCIe switch for completion [49]. The latency of SEND/RECV on SNIC also increases, but mainly due to the larger CPU costs at the responder; the latency to post a request (via MMIO) on SNIC is higher than RNIC (399 cycles vs. 279 cycles).

**Throughput.** As shown in Figure 3, for READ, WRITE, and SEND/RECV, SNIC ① has 19–26%, 15–22%, and 3–36% lower throughput than RNIC ① for payloads less than 512 bytes, respectively. We suspect the lower throughput is due to the longer latency in processing RDMA requests caused by PCIe switch. However, for larger requests, the results are similar to using RNIC as both are bottlenecked by the network bandwidth.



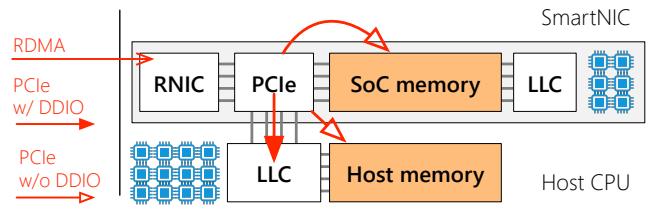
**Figure 5:** (a) An illustration of three examples of different combinations of data flows on different paths, and (b) the peak throughput of different combinations of data flows on different communications paths.

**Bottleneck.** The lowest bandwidth limit of NIC, PCIe1, and PCIe0 will first become the bottleneck for communication from client to host. On our testbed, the bottleneck is the network: 200 Gbps. On the other hand, we find an interesting phenomenon: the total inbound bandwidth of the requester can approach *twice* the limit—400 Gbps—because the links are *bi-directional* [58]. Specifically, if packets flow in opposite directions, e.g., the READ and WRITE packets in Figure 5(a), they can be multiplexed on the same link. To illustrate this, we dedicate two requesters (each with 12 threads to saturate the one-way bandwidth) to issue 4 KB packets. As shown in Figure 5(b), if two clients send READ and WRITE requests separately, a total of 364 Gbps bandwidth is measured on a 200 Gbps NIC (see READ+WRITE of SNIC ①). In contrast, if both clients send the same type of requests (either READ or WRITE), only about 190 Gbps is measured. Note that though this phenomenon is widely known in traditional networking (i.e., messaging), where the messages are typically two-sided, it is largely ignored by many RDMA-based systems, because RDMA request can be one-sided.

**Takeaways.** Being “smart” incurs performance degradation for communicating with the host for small requests. For small requests, we demonstrate that extending RNIC (ConnectX-6) to SNIC (Bluefield-2) causes performance degradation by up to 36% and 30% in throughput and latency, respectively. In general, for distributed systems that only use the path ①, it is recommended to use RNIC. Although the overhead may be negligible for large requests or for networking with longer latency, RNIC is cheaper and more energy-efficient than SNIC.

### 3.2 Communication from Client to SoC (path ②)

**Latency.** For sending requests from the client to SoC (SNIC ② in Figure 3), the latency of READ decreases by up to 14% compared to the host (SNIC ①). The *reason* is that it skips PCIe0. Yet, it is still 4–15% higher than RNIC, because requests still must go through the PCIe switch at PCIe1. For WRITE, SNIC ② provides similar performance as SNIC ① due to the asynchronous completion of cores (see Figure 4). For SEND/RECV, SNIC ② has 21–30% higher latency than SNIC ① due to the weaker computing power of SoC.



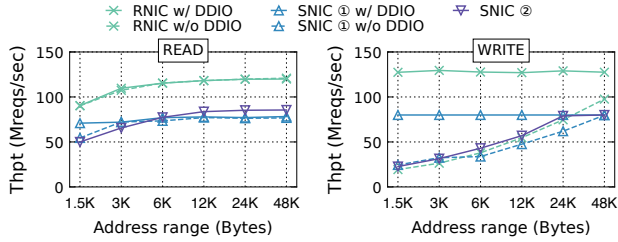
**Figure 6:** Different paths to access host and SoC memories.

**Throughput.** SNIC ② has better throughput than SNIC ①, reaching  $1.08\text{--}1.48\times$  for payloads less than 512 bytes. Interestingly, the READ of SNIC ② is even higher than that of RNIC ① before reaching the peak network bandwidth. For this undocumented results, we suspect that it is due to the closer packaging of SoC memory and the PCIe switch. Specifically, the SoC is linked to the PCIe switch via an internal link, rather than through PCIe. Note that a confident analysis relies on the hardware details of Bluefield, which unfortunately are not available now. For WRITE, SNIC ② is still lower than that of the RNIC ①. Our *hypotheses* are twofolds. First, SoC has fewer DRAM channels compared to the host (1 vs. 4), limiting the concurrency of write accesses. Nevertheless, READ is not affected because read accesses on DRAM are faster than write accesses [25, 73]. Second, SoC can only utilize a portion of NIC cores (see §4.1). Finally, SEND/RECV has a poor performance on ②: it just achieves up to 64% of the host (SNIC ①). This is due to the wimpy computing power of SoC, since the throughput of SEND/RECV is bottlenecked by the responder CPU to send the reply.

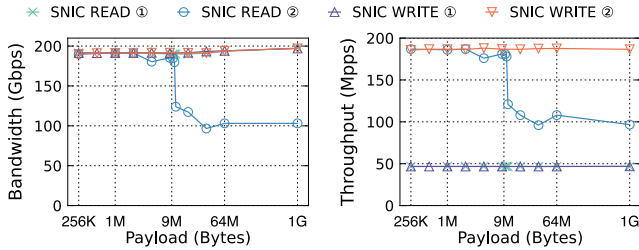
**Bottleneck.** As shown in Figure 1(c), since SNIC ② only flows through NIC and PCIe1, the bottleneck is their lower bandwidth limit, which is still Bluefield-2’s 200 Gbps NIC. Therefore, as shown in Figure 5(b), the performance of SNIC ② is the same as that of SNIC ①, namely the total of 400 Gbps and 200 Gbps bandwidth for opposite direction and same direction communication, respectively.

In addition to the basic RDMA performance of the SNIC, we found several factors that could also prevent distributed systems from achieving the aforementioned performance.

**Advice #1: Avoid memory accesses to close addresses.** The



**Figure 7:** The peak throughput of accessing host memory and SoC memory via SNIC, READ (a) and WRITE (b).



**Figure 8:** The bandwidth (a) and PCIe packet throughput (b) for accessing (READ and WRITE) the host (SNIC ①) and SoC (SNIC ②) via SmartNIC. For brevity, we omit the result of SEND/RECV since it is the same as WRITE for large payloads [27].

wimpy SoC cores may impact the memory access behavior of one-sided RDMA primitives, because it usually supports fewer features compared to the more powerful host CPU cores. Specifically, Data Direct I/O (DDIO) [26] is widely supported by the host CPUs, which allows the NIC to directly read/write data from/to its last level cache (LLC), as shown in Figure 6. SoC cores may also equip with similar features (e.g., ARM CCI [5]), but whether to do so is vendor-specific. The SoC cores of our hardware (ARM Cortex-A72 in Bluefield-2) do not support DDIO. We find that one-sided RDMA without DDIO suffers performance drop if the requested memory addresses fall into a small range (i.e., they are close together). This is *because* DRAM requires a (not-too-small range) to utilize all memory modules concurrently. LLC is faster than DRAM, so we suspect the impact is smaller.

Figure 7 shows the peak throughput of accessing host memory and SoC memory via SNIC with the increase of address ranges.<sup>5</sup> For WRITE, the throughput of SNIC ② using SoC drops to 22.7 M reqs/s (from 77.9 M reqs/s) when address range decreases to 1.5 KB (from 48 KB). In contrast, the performance of SNIC ① using Host CPU is hardly affected when DDIO is enabled. For READ, the degradation is relatively smaller. The throughput of SNIC ② drops from 85 M reqs/s to 50 M reqs/s when decreasing the range from 48 KB to 1.5 KB. This is because DRAM can serve reads faster than writes [25, 73]. Finally, we also plot the RNIC results as a reference. When requests addresses are close, we can see that ① also suffers a significant performance drop on WRITE when DDIO is disabled.

<sup>5</sup>Note that we attach Bluefield to CLI machines for the evaluation because we are unable to disable DDIO on the SRV machines.

**Table 4:** PCIe Maximum Transfer Unit (MTU) on our testbed, and the number of PCIe packets required to transfer  $N$  bytes via different communication paths of Bluefield-2. Our simplified model omits control-path packets (e.g., two-sided message arrival notification).

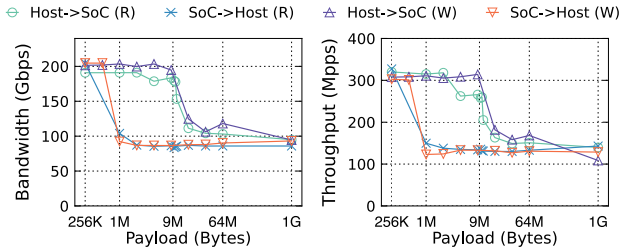
	Host CPU cores ( $H_{MTU}$ )		SoC cores ( $S_{MTU}$ )
PCIe MTU	512 B		128 B
	SNIC ①	SNIC ②	SNIC ③
PCIe1	$\lceil N/H_{MTU} \rceil$	$\lceil N/S_{MTU} \rceil$	$\lceil N/H_{MTU} \rceil + \lceil N/S_{MTU} \rceil$
PCIe0	$\lceil N/H_{MTU} \rceil$	–	$\lceil N/H_{MTU} \rceil$

**Advice #2: Avoid large READ requests.** It is common practice to use requests with large payloads to fully exploit network bandwidth. For example, using requests with payloads larger than 16 KB is enough to saturate a 200 Gbps RNIC even using a few threads. Unfortunately, we observed that the READ performance of SNIC ② collapses with request payload larger than 9 MB, as shown in Figure 8(a). We *suspect* that NIC cores suffer from head-of-line blocking when processing large READ requests. For a READ request, the NIC issues a PCIe read transaction to fetch the data, which is further segmented into multiple PCIe packets. The maximum size of a PCIe packet is determined by the PCIe Maximum Transfer Unit (MTU), negotiated by the linked hardware devices during bootstrap [49]. Table 4 lists the PCIe MTU on our testbed. SoC cores (the endpoint of SNIC ②) use a smaller PCIe MTU (128 B) due to its weaker CPU. As a result, NIC core that processes a large DMA read sent to SoC memory (SNIC ②) must wait for more PCIe packets to arrive, resulting in lengthy processing stalls. Since the overall NIC packet processing power is not the bottleneck: as shown in Figure 8(b), the requests with payloads smaller than 9 MB still can achieve a high processing rate while it collapses for the others, so we suspect some blocking happens at the NIC core. Note that WRITE requests are not affected since DMA does not wait for the completion [83, 49].

On the contrary, the host uses a larger PCIe MTU (512 B), so it does not suffer from bandwidth degradation (SNIC ①). As shown in Figure 8(b), the NIC can issue 46.7 million PCIe packets per second to the host (SNIC ①). The aggregated bandwidth reaches 191 Gbps, bottlenecked by the network.

**Takeaways.** For READ and WRITE, sending requests to SoC is typically faster than that to the host (or even faster than via RNIC) because SoC is “closer” to the NIC (without PCIe0). In contrast, using SEND/RECV to communicate with SoC is slower due to weak SoC cores. Furthermore, designers still need to carefully consider the heterogeneity between host CPU cores and SoC cores to avoid performance anomalies. Specifically, memory accesses to a small address range may suffer performance degradation due to the lack of DDIO support on SoC cores. In addition, sending large READ requests to SoC may underutilize the bandwidth so the request should be proactively segmented into smaller ones.





**Figure 9:** The bandwidth (a) and PCIe packets throughput (b) for sending (R)EAD/(W)RITE requests between the host and SoC.

### 3.3 Communication between SoC and Host (path ③)

We first describe our measurements and findings of RDMA and then compare RDMA (③) with DMA (③\*).

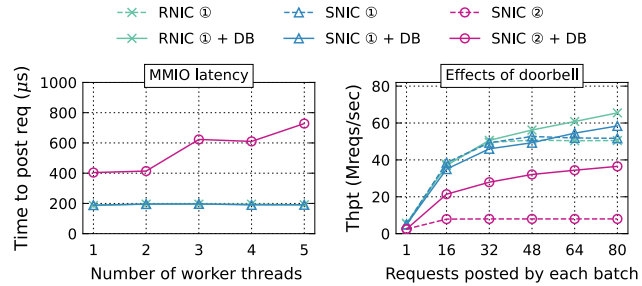
**Latency.** As shown in Figure 3, the latency of sending requests from SoC to the host (SNIC ③ S2H) is very high, especially for READ, since the requester (SoC) takes longer to issue an RDMA request to the NIC. The latency in the opposite direction (from the host to SoC, SNIC ③ H2S) is reduced but still 4–17% higher than SNIC ②. Although the intra-machine communication saves one network round-trip, it adds additional PCIe transfers. Specifically, the request on SNIC ② flows the requester-side PCIe (not shown in Figure 1(c)), the network, PCIe1, and the PCIe switch, while the request on SNIC ③ (H2S) flows PCIe0, the PCIe switch, PCIe1 twice (in and out), and the PCI switch (again).

**Throughput.** For requests with payloads less than 512 bytes, the throughput of SNIC ③ (both S2H and H2S) is dominated by the requester’s capability to post networking requests. This is because a single requester machine (either SoC or the host) cannot saturate the NIC with small requests<sup>6</sup>, the READ throughput of SNIC ③ only reaches 29 M reqs/s and 51.2 M reqs/s for S2H and H2S, respectively, still far from its limit. For WRITE and SEND/RECV, the results are similar. For larger requests, they are bottlenecked by the PCIe bandwidth, which will be discussed in more detail next.

**Bottleneck.** As shown in Figure 5(b), for packets flowing in a single direction, communication between host and SoC is bottlenecked by PCIe bandwidth (256 Gbps) rather than the uninvolved NIC (200 Gbps). Therefore, the peak bandwidth of SNIC ③ is slightly higher than SNIC ① and ② (204 Gbps vs. 191 Gbps). Readers might be interested in why the results of SNIC ③ cannot be close to 256 Gbps. We suspect that it requires much more PCIe packets than the others. For packets flowing in opposite directions, SNIC ③ can not utilize twice the limit as the other paths (i.e., SNIC ① and ②). This is because RDMA overuses the PCIe: each request passes through PCIe1 twice (in and out), exhausting the bi-directional link.

**Advice #3: Avoid large READ/WRTIE requests.** Communications between the host and SoC (SNIC ③) also suffers from bandwidth degradation for large READ requests like SNIC ②,

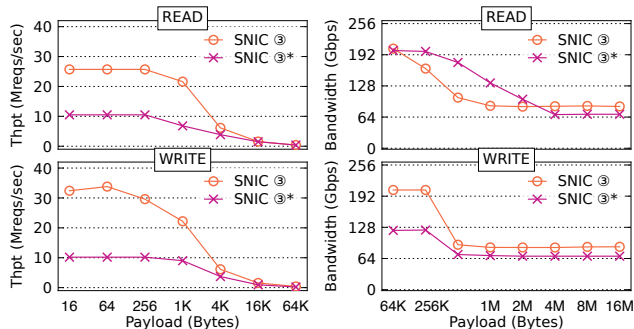
<sup>6</sup>We use up to eleven requester machines for SNIC ① and SNIC ②.



**Figure 10:** The latency of posting requests to NICs (a) and the impact of doorbell batching (DB) on the requester.

possibly due to the head of line blocking as we have discussed before. Moreover, this issue appears with large WRITE requests because the SmartNIC must first read data from the requester and then write it to the responder. As shown in Figure 9(a), the READ/WRITE performance of SNIC ③ collapses to about 100 Gbps for large requests. Table 4 shows the number of PCIe packets required to transfer  $N$  bytes via different communication paths. For SNIC ③, the NIC generates more packets due to passing through PCIe1 twice. Further, the performance of S2H collapses earlier than H2S as it will pass through PCIe1 first. Suppose we transfer data at 200 Gbps from SoC to the host. The SoC cores first transfer 195 M PCIe packets per second (pps) to the NIC (PCIe1), then the NIC forwards data back to the PCIe switch via PCIe1 again with 49 Mpps (the host supports 512 B MTU), and finally, the switch forwards 49 Mpps through PCIe0. Therefore, SmartNIC should process at least 293 Mpps for transferring data at 200 Gbps, which is  $3\times$  and  $1.5\times$  higher than SNIC ① and SNIC ②, respectively. This is further confirmed by our measurements of the hardware counters. As shown in Figure 9(b), for sending 256 KB READ requests from SoC to the host, the bandwidth reaches 204 Gbps, and the NIC transfers about 320 M PCIe packets per second.

**Advice #4: Enable doorbell batching carefully.** The time of posting each request to the NIC is dominated by Memory-Mapped IO (MMIO) [76, 28]. The SoC suffers a high MMIO latency when communicating with the host (see Figure 10(a)). A known optimization is doorbell batching (DB) [28]: to send a batch of  $B$  requests, the requester first chains them together in memory, then use one MMIO to ask the NIC to read these requests with DMA in a CPU-bypass way. DB reduces the number of MMIOs required from  $B$  to 1. Thus, for RNIC ① and SNIC ②, DB is always helpful and can bring 2–30% performance improvement (see Figure 10(b)). For the communication between host and SoC (SNIC ③), DB is still helpful at the SoC-side. As shown in Figure 10(b), when sending a batch of READs to the host, DB improves the SoC performance by 2.7–4.6 $\times$  for batch sizes 16–80. The huge improvement is partly due to the CPU-bypass feature of DMA, and also because the NIC is faster in using DMA to read requests stored on SoC memory (see §3.2). However, DB is not always helpful at the host-side, because it is slower to



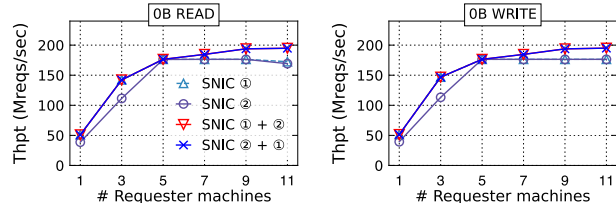
**Figure 11:** The throughput and bandwidth comparisons of using RDMA (③) and DMA (③\*) when communicating from SoC to the host, the only supported primitive of DMA of our SNIC.

read host memory using NIC DMA (see §3.1). For batch sizes of 16, 32, and 48, DB decreases the throughput of host-SoC communication by 9%, 7%, and 6%, respectively.

**RDMA (③) vs. DMA (③\*).** Besides RDMA, SoC can use DMA (③\*) to read/write data from the host (and vice versa) via the DMA engine inside the SoC. It has the benefits of reducing two PCIe passes (PCIe1) and bypassing RNIC compared to RDMA (see Figure 1), resulting in a lower latency, e.g.,  $1.9 \mu$  vs.  $2.6 \mu$  for 64 B SoC to host READ. However, we find the SoC DMA engine has a weaker processing power than RNIC (RDMA). For brevity, we only present results on SoC to host. The results of host to SoC is the same as SoC to host since host DMAs are offloaded to SoC for execution [56]. As shown in Figure 11, for WRITE, the peak throughput of DMA is only 47–59% of that of RDMA for requests with payload less than 4 KB. The results of READ is similar. DMA WRITE even fails to saturate the PCIe limit (256 Gbps) for payloads between 16 KB and 1 MB. We suspect it is due to the poor processing capability of the SoC’s DMA engine, yet we cannot confirm this without knowing the confidential internal design of the SoC. Another observation from the bandwidth results is that DMA also suffers from the anomalies of RDMA (see Advice #3): For payloads larger than 1 MB, there is a significant performance drop for both READ and WRITE.

For bandwidth, ③\* has a higher theoretical upper bound than ③: it is bottlenecked by the bidirectional bandwidth of PCIe, as it bypasses the PCIe1. However, Figure 5 shows that it fails to achieve so (only 178 Gbps for READ + WRITE). This suggests that the slow DMA engine will first become the bottleneck. Nevertheless, bypassing PCIe1 still has the benefits of reducing interferences to other paths. We will discuss them in §4 in detail.

**Takeaways.** First, enabling doorbell batching is critical for SNIC ③ at the SoC side, because SoC has wimpy computation power. Yet, it is negatively impacted at the host side for small batch sizes. Second, SNIC ③ has a different bottleneck than SNIC ① and SNIC ②. It is always bottlenecked by the uni-directional bandwidth of PCIe, while others are limited by the minimal bi-directional bandwidth of network and PCIe.



**Figure 12:** Throughput for (a) READ and (b) WRITE with the increases of requester machines.

If this factor is not adequately considered, distributed systems will underutilize the NIC bandwidth (see §5.1). Third, though DMA utilizes PCIe better than RDMA for SoC to communicate with the host, it has a lower throughput due to the weaker DMA engine at the SoC. Finally, we should avoid transferring large requests between the host and SoC, for both RDMA and DMA and for both READ and WRITE.

## 4 A Guideline for Smartly Exploiting Multiple Paths of SmartNIC

Previous approaches mainly leverage a single path of SmartNIC to optimize a specific functionality of distributed systems. However, this cannot fully exploit the computing and networking capabilities of SmartNICs. Furthermore, only considering a single path may ignore interference on resources (e.g., PCIe and PCIe switch) between different paths. Therefore, we first holistically study the performance characteristics of concurrently using multiple paths, and then lay out an optimization guideline for designers to smartly use SmartNICs.

### 4.1 Characterizing concurrent communication paths

**Concurrent communication with the host and the SoC (①+②).** We focus on the throughput results (see the lower part of Figure 3) since the latency results are roughly the average of the two paths. We evaluate the peak throughput by assigning half of the clients to send requests to the host while the others to send to the SoC. We can see that the total peak throughput of concurrently using ① and ② (SNIC ①+②) is typically faster than each of them. For READ, WRITE, and SEND/RECV, SNIC ①+② outperforms the lower of them by up to  $1.45\times$ ,  $1.50\times$ , and  $3.3\times$ , respectively.

For SEND/RECV, a concurrent path utilize both of the host and SoC to process the requests, so the performance improvement is clear. However, the READ/WRITE performance improvement is non-intuitive and undocumented, since two paths should compete for NIC cores. Our *suspicion* is that the SmartNIC internally reserves some NIC cores for each endpoint. Therefore, sending requests to the host and the SoC concurrently can further increase *peak* throughput by enabling more NIC cores. To quantify this, we design a microbenchmark that first increases the requester machines to saturate the NIC and then changes the responder, as shown in Figure 12. All requests use 0 B payload to avoid interference of DMA, i.e., the request will return before passing PCIe1 [6].

For READ, five requester machines are sufficient to saturate NIC cores when using SNIC ① or SNIC ② alone. Therefore, for concurrently using SNIC ① and SNIC ②, we first dedicate five requester machines for one responder, and then add requesters for the other responder. Both cases (SNIC ①+② and SNIC ②+①) offer similar performance, with 4–13% and 5–10% higher throughput than using SNIC ① or SNIC ② alone. For WRITE, all results are almost the same.

Finally, as expected, the aggregated throughput of the two paths (SNIC ① and SNIC ②) is much higher than concurrently using them (352 Mpps vs. 195 Mpps), indicating that most NIC cores are still shared, i.e., each can communicate with two endpoints, and only a few is dedicated. This also implies that concurrently using multiple resources of SmartNIC is non-trivial.

**Concurrent inter- and intra-machine communication (①/②+③).** There exist four concurrent combinations of inter- and intra-machine communication. For brevity, we focus on the results of SNIC ①+③/H2S, other combinations are similar. To study the concurrent usage of the two paths, we first deploy sufficient clients (five requester machines) to saturate the network for SNIC ①. Afterward, we start the requester on the host (one machine with 24 threads) sending RDMA requests to the SoC (SNIC ③/H2S). Our measurements reveal that concurrently enabling intra-machine communication degrades the performance of inter-machine communication. As shown in Figure 3, for READ, WRITE, and SEND/RECV, the throughput of small requests (less than 512 bytes) drops 7–15%, 4–27%, and 9–14%, by comparing SNIC ① and SNIC ①+③(H2S). For large requests, the performance is always bottlenecked by the network bandwidth, so the degradation is negligible.

The SNIC ③ affects other communication paths of SmartNIC, because it relies on the NIC (PCIe1 and the PCIe switch) for RDMA support. In comparison, SNIC ③\* communication can leverage DMA to reduce such interferences. For example, for READ with payloads 16–64 B, we only observe a 5–6% throughput drop, after adding SNIC ③\* to ①.

**Bottleneck.** Assuming each path has only one type of request, e.g., either READ or WRITE. For SNIC ①+②, each part has the same bottleneck (the NIC), so the bandwidth limit is 400 Gbps (bi-directional). For SNIC ①+③, it is bottlenecked by SNIC ③, which is limited on the uni-direction of PCIe (256 Gbps) since it occupies both directions of PCIe1 (see Figure 5(b)). Nevertheless, if SNIC ① is used in opposite directions (i.e., READ and WRITE), SNIC ①+③ can reach a higher limit. For example, the aggregated bandwidth can achieve 456 Gbps (in theory) if we restrict the bandwidth of data transfer on SNIC ③ to 56 Gbps. This suggests that selectively offloading small portion of data to SoC may be optimal. Finally, if possible, it is usually better to combine SNIC ① or ② with DMA (①/②+③\*) despite DMA being slower than RDMA (see §3.3). This is because DMA has

better PCIe utilization (without passing PCIe) and RNIC utilization (without using RNIC).

**Takeaways.** Sending requests from clients to the host and the SoC concurrently (SNIC ①+②) can better utilize NIC cores to handle small RDMA requests, especially when used in opposition directions (e.g., one for READ and one for WRITE). On the contrary, uncontrolled use of intra-machine (host-SoC) communications (SNIC ③) may harm inter-machine communications, which is the intrinsic purpose of using SmartNIC. Specifically, if the uni-directional bandwidth of PCIe is smaller than the bi-directional bandwidth of the NIC, using SNIC ③ can introduce a hidden bottleneck. Therefore, we should always consider using SNIC ③ only when spare resources are made available. Specifically, if the inter-machine communication saturates the NIC, the bandwidth used by SNIC ③ should no larger than  $P - N$ , where  $P$  and  $N$  are the limit of the PCIe and the network, respectively. For example, it should be 56 Gbps on our testbed. Using SNIC ③\* can reduce the interference between paths, but SNIC ③\* also has limitations: it is slower than SNIC ③.

Finally, in real-world distributed systems, it is common that a single communication path cannot fully saturate all resources of SmartNIC. For example, SNIC ② is the fastest but limited by small memory and wimpy cores on the SoC. On the other hand, only using SNIC ① as RNIC would waste all resources on the SoC. Therefore, we should concurrently use multiple paths provided by the SmartNIC, but carefully avoid interference between them.

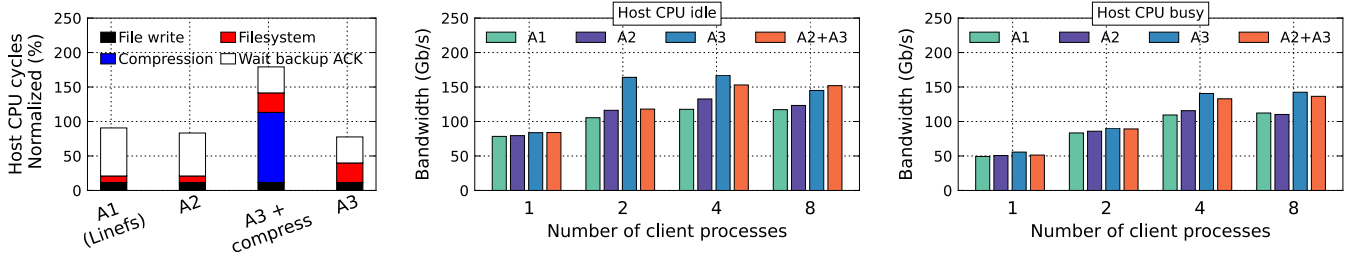
## 4.2 An optimization guideline

This section presents our optimization guideline for smartly utilizing multiple communication paths of SmartNIC to improve the performance of distributed systems. Specifically, given the functionality (e.g., file replication in a distributed file system) of a target distributed system that needs to be accelerated by SmartNIC, we recommend designers consider the following steps:

1. Devise potential alternatives for SmartNIC to support the given functionality, and optimize them based on performance characteristics uncovered by our study.
2. Evaluate and rank alternatives based on system-specific criteria.
3. Select and combine alternatives in turn until the resource of SmartNIC is saturated.

**System-specific criteria.** The criteria can be the desirable properties that the system designer aims to achieve, or the restrictions of the systems. For replication in a distributed file system, the properties include low host CPU overhead and high network bandwidth utilization [32]. For a disaggregated key-value store, the properties include less network amplification, low latency and high throughput. The restriction the host has little or no CPU that we can use [86].





**Figure 13:** The host CPU usage breakdown of different alternatives when replicate 8 MB data (a). Write bandwidth when the host is idle (b) and busy (c) for A1, A2 and A2 + A3.

**Discussion.** We currently only consider the combination of alternatives in a greedy way, which is sufficient for most networked functions in real-world distributed systems. Further, SmartNIC usually offers a limited number of available options. Note that efficiently combining alternatives is challenging. For different systems, different alternatives may consume different resources on the SmartNIC, while a combination of them may involve different levels of resource contentions. Our previous analysis—including the bottleneck of different communication paths and concurrently utilizing multiple paths on the SmartNIC—will guide designers to avoid most performance contention. Nevertheless, how to systematically choose and combine different paths is our future work.

## 5 Case Studies

To demonstrate the efficacy of our study and the optimization guideline, this section presents two detailed case studies.

### 5.1 Distributed file system

**Overview.** File replication is a key pillar in distributed file systems for fault tolerance. With the emergence of RDMA and non-volatile memory (NVM), an appealing trend is to use RDMA to directly replicate file updates on remote NVM for better performance [32, 3, 40, 4], i.e., RDMA primitives can directly write NVM just like DRAM, with network and NVM bandwidth fully utilized [81].

**Devise alternatives.** The desirable properties of file replication are high performance, high network utilization and low host CPU overhead. There are three alternatives to implement file replications on our SmartNIC, as illustrated in Figure 14.

1. **Alternative (A1).** It comes from the state-of-the-art distributed file system on SmartNIC, LineFS [32], which completely offloads the file replication to SoC. The SoC will compress and replicate the file to reduce data transferred through the network with low host CPU usage. After receiving a replication request, the primary SoC reads the file from host (③), compresses it (④), and writes the file to remote backups with chain replication [72] (②). Specifically, if there are multiple backups, the second backup will further re-replicate the log to the next backup on the chain and so on.
2. **Alternative (A2).** Guided by our study, we can replace

the ③ in A1 with ③\* to reduce interference on the PCIe bandwidth, specifically, PCIe1 on the SmartNIC.

3. **Alternative (A3).** The host can directly write the file from the host to the remote backup with WRITE (①) [40]. Note that this approach typically skips file compression to prevent non-trivial host CPU overhead (see Figure 13 (a)).

**Baseline.** LineFS [32] is a state-of-the-art distributed file system based on NVM and SmartNIC. It adopts A1 to replicate the files. We further implement A2 and A3 on its open-source codebase<sup>7</sup>, and rewrite its backend with more efficient RDMA implementation to scale to 200 Gbps networking, e.g., with asynchronous and batched RDMA operations.

**Optimization on each alternative.** By default, LineFS adopts a chunk size of 16 MB in its open-source codebase for A1. Based on our Advice #3 described in §3.3, we shrink it to 256 KB for a better performance over ③. This optimization further applies to A2 and A3.

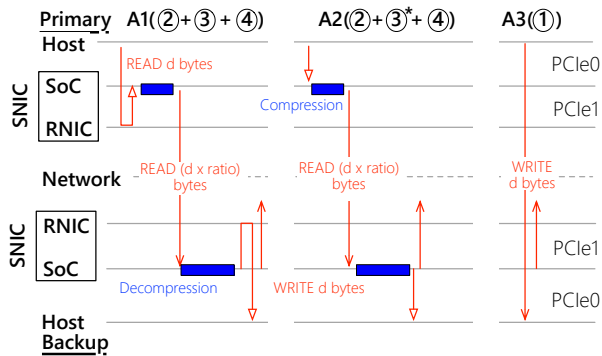
**Analyse alternatives.** A1 is the most straightforward way to offload file replication, reducing the data transferred through the network ( $d$  vs.  $d \times ratio$ ). Thus, the ideal peak bandwidth is  $N/ratio$ , where  $N$  is the bandwidth limit of SmartNIC. However, A1 does not consider the costly PCIe occupation of ③ (§3.3), which even fails to saturate the network bandwidth for file transfer. Denote the primary’s PCIe limit (uni) as  $P$ . A1’s file transfer bandwidth  $d$  is limited by  $\frac{P}{1+ratio}$ , because each data packet must pass the *PCIe1 out link* twice. As shown in Figure 14, one is from SoC to RNIC ( $d$  bytes) and another from SoC to the remote ( $d \times ratio$  bytes). On our platform ( $p = 256 Gbps$ ), so A1 is only better than file is not compressed (whose performance is bottlenecked by the network  $N = 200 Gbps$ ) when the compression ratio is lower than 28%. Worse even, A1 cannot saturate the network bandwidth of SmartNIC when encountering a bad compression ratio ( $\geq 28\%$ ). For example, without compression ( $ratio = 1$ ), the peak of A1 is only 128 Gbps.

Figure 13 (b) presents the results of A1 on the file write benchmark of LineFS. This benchmark does not compress the file. We can see that A1 only achieves 117 Gbps with 8 clients when the host is idle.

A2 addresses the poor PCIe utilization of A1 by replacing ③ with ③\*. As shown in Figure 13 (b), A2 is 1.01–1.13 ×

<sup>7</sup><https://github.com/casys-kaist/LineFS>





**Figure 14:** Overview of alternatives for file replication with SmartNIC. *ratio* is defined as `compressed size / uncompressed size`. We omit the control path messages as they are trivial.

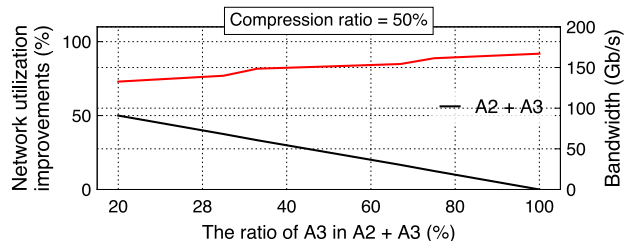
faster than A1 under different number of clients. However, A2 fails to achieve a close to 200 Gbps result (peak at 133 Gbps) due to the following two reasons. First, the WRITE of ③\* cannot fully utilize the full PCIe bandwidth on our platform (see Figure 11). Second, the poor computation power of SoC may also become the performance bottleneck of file replication.

A3 bypasses the PCIe occupation problem of A1, and the slow DMA WRITE and weak SoC issues of A2. Meanwhile, its data path is shorter (see Figure 14). As shown in Figure 13(a), it takes 40% shorter time to wait for the log acknowledgment compared to A2. As a result, A3’s replication bandwidth is 5–41% faster than A2 under different client setups. The drawback is that A3 takes more CPU cycles even without considering compression (Filesystem), see Figure 13(a). This is because A1 and A2 can digest the file log on the SoC. Thus, the overall process time reduction of A3 is 8% (decreased from 40%) compared to A2.

**Select and combine alternatives.** Since A2 is always better than A1, we will only consider combining A2 with A3. As we have analyzed before, A3 is faster than A2. Therefore, increasing the ratio of A3 in a combined path (A2 + A3) always improves the performance, as shown in Figure 15. However, if file compression for high network utilization is enabled, it has high host CPU utilization, as shown in Figure 13 (a). Disabling compression for A3 will lower the network utilization, also illustrated in Figure 15. Specifically, when increasing the percentage of path A3 in clients, the network utilization is reduced from 50% to 0% considering a fixed 50% compression ratio.

Considering A2 has better network utilization, we follow a greedy approach that first saturate the SoC with A2 for better network utilization. Afterward, clients use A3 to do the file replication. This approach can achieve the best of both worlds: the combined path is faster than A2 with network better utilized than A3.

**Evaluation results.** Figure 13 (b) and (c) further present the file replication benchmark results of A2 + A3 when the host CPU is idle and busy, respectively. We follow the same setup as LineFS [32]’s benchmark and add a CPU-intensive



**Figure 15:** Analysis of the network utilization and performance when combining A2 and A3.

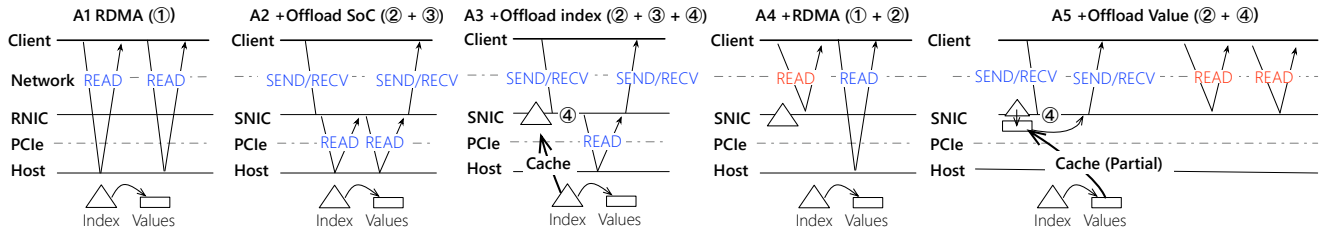
workload (streamcluster [7]) to the host CPU to emulate a busy experimental setup. A2 + A3 is 7–30% and 4–21% faster than original LineFS when CPU is idle and busy, respectively, thanks to the more efficient usage of SmartNIC and a smart utilization of multiple execution paths.

## 5.2 Disaggregated key-value store

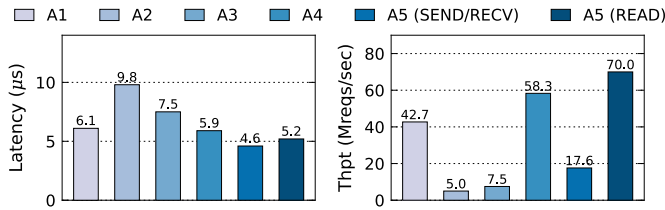
**Overview.** RDMA-based disaggregated key-value stores (R-KVS) are prevalent in modern data centers [75, 70, 17, 86]. In R-KVS, one or more memory servers store both indexes (usually hash table) and values. Clients on other machines use READs to traverse the index and retrieve the corresponding value to handle requests (i.e., get), see A1 in Figure 16.

**Devise alternatives.** The desired properties are high throughput, low latency and minimal network amplification. The restriction is that we can barely use the host CPU (i.e., disable SEND/RECV for path ①). SmartNIC enables five alternatives for R-KVS, as illustrated in Figure 16.

1. **Alternative (A1).** The client treats SmartNIC as a normal RNIC and uses READs to handle the get request (①). This approach suffers from network amplification.
2. **Alternative (A2).** One intuitive approach for offloading is to send the get request to the SoC using SEND/RECV (②). The SoC can then traverse the index and read the value on the host via RDMA or DMA READ. This approach effectively eliminates network amplification.
3. **Alternative (A3).** One drawback of A2 is that reading data from SoC to the host is slower reading from the host’s local memory. An optimization is to offload the indexes to the SoC memory (④). This approach is similar to index caching at the clients [11, 62, 75], but caching the indexes at the SmartNIC is more effective. Each client has a small memory that can only cache hundreds of entries in a disaggregated setting [86], while SmartNIC has a relatively large SoC memory (e.g., 16 GB on Bluefield-2) that can cache all the indexes.
4. **Alternative (A4).** Accessing the index on the SoC using SEND/RECV (②) cannot fully utilize the NIC cores of SmartNIC, because the peak throughput of SEND/RECV is only 21.6 M reqs/s. Therefore, we can use READs to traverse the index on the SoC (②), and another READ to retrieve the value on the host (①).



**Figure 16:** Alternatives (A1–A5) for offloading a get request of RDMA-based disaggregated key-value store to off-path SmartNIC.



**Figure 17:** (a) Latency and (b) throughput comparisons between different alternatives on YCSB C. For A5, we restricted the selection of clients keys to always hit the cached values on SmartNIC.

This approach still has network amplification, but can utilize the fast path (②) to improve performance (see §3.2).

- Alternative (A5).** Similar to index caching, SoC memory can further cache a portion of values (e.g., the values of hot keys). This approach avoids using the costly communication path (③) of the previous alternatives.

**Baseline.** DrTM-KV [11] is a state-of-the-art KV store optimized for RDMA: it adopts cluster-chaining hash index such that the client typically finds the value position of a given key in one READ. Specifically, for a get request, the client first READs a 64 B bucket (based on the hash of the key), finds the remote address of the corresponding value in it, and then fetches the value with another READ. DrTM-KV supports index caching at the client to skip the first READ [80], but it may not be always feasible in a disaggregated environment due to memory constraints [86], so we disable it.

**Optimization on each alternative.** We implement A1–A5 on DrTM-KV guided by our study (§3). Specifically, we carefully enabled doorbell batching for alternatives related to the SoC CPU (A2, A3 and A5). Besides, we apply Advice #1 for A4 and A5, which replicate a few hot keys to multiple replications to avoid sending requests to a small range of memory. We use DMA (③\*) instead of RDMA (③) to implement A2 and A3 as it is always faster due to lower latency. For example, A2 throughput is improved by up to 79% with ③\*. A2 and A3 does not suffer from the low DMA throughput discovered in §3.3 because the SoC will first become the bottleneck.

**Analyse alternatives.** We use YCSB C [16] (100% get) with default Zipfian request distribution ( $\theta = 0.99$ ) for all the experiments. The payload sizes of keys and values are 8 B

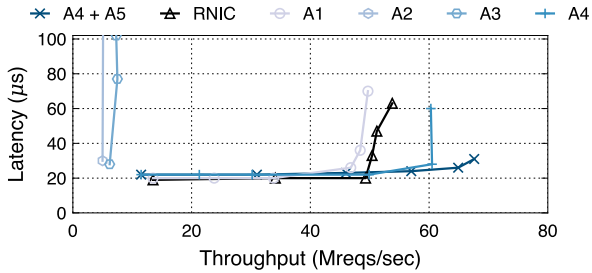
and 64 B, respectively, similar to prior work [41, 45, 30, 66, 75]. Following the microbenchmark setup, we use one client machine to measure the latency and deploy up to eleven client machines to measure the peak throughput.

Figure 17 demonstrates that none of the path can achieve both high throughput and low latency. A5 (SEND/RCV) achieves the lowest latency ( $4.6 \mu s$ ) because it completely eliminates the network amplifications problem and costly host-SoC communications (③). However, its peak throughput (17.6 M reqs/s) is significantly lower than some other alternatives. Specifically, the peak throughput of A5 (READ) and A4 reach 70 M reqs/s and 58.3 M reqs/s, respectively. They have a higher throughput because the RDMA path to SoC (①) is faster (§3.2). Note that A5 is not always achievable, which requires caching all the key-values at the SoC memory. Therefore, A4 is a suitable design if the SoC cores become the bottleneck (④). A1 has a higher latency and lower throughput than A4, since RDMA to the host (①) is relatively slow. A2 and A3 are bottlenecked by the slow host-SoC communication (③, see §3.3), which is not suitable for offloading KV store requests.

**Select and combine alternatives.** Our analysis suggests that the optimal combination is A4 and A5. Initially, the first few clients use A5, whereas the later clients use A4. The exact switch point can be estimated by using queuing theory [24] to model the capacity of SoC and the capability of RNIC, as in prior work [46].

In addition, using A5 presents a challenge as clients are unaware of which values are cached at SoC. Although A3 can be used as a fallback path for cache misses, it will result in significant performance degradation (see Figure 17). To tackle this issue, we provide a simple solution: when a cache miss occurs, the SoC returns the address of the value to the client, which then issues a READ to retrieve the value accordingly, similar to A4. In real-world skewed workloads (e.g., YCSB [16]), cache misses are rare.

**Evaluation results.** Figure 18 shows the latency and throughput results on YCSB C. We plotted the graph by increasing the number of client machines. The combination of A4 + A5 achieves a peak throughput of 68 M reqs/s, which is 25%, 36%, and 12% higher than RNIC, A1, and A4, respectively. Note that we omit A2 and A3 as they are bottlenecked by SoC cores and have extremely low peak throughput. The benefits



**Figure 18:** Performance of YCSB C using different alternatives. Note that A5 cannot run a full workload alone, since SoC memory is not large enough to cache all values. For A4 + A5, one client uses A5, and the rest use A4.

of A4 + A5 mainly come from utilizing faster SoC RDMA and SoC cores for reducing network amplifications.

## 6 Discussion

**Generalizability.** Although our study primarily focuses on one particular SmartNIC, Bluefield-2 [52], we believe that our findings and advice can be applied to other off-path SmartNICs that share a similar hardware architecture. These SmartNICs extend RDMA-capable NICs, such as Stingray PS225 [9] (which extends NetXtreme 100 Gbps RNIC [8]), by attaching a heterogeneous SoC and bridging SoC and RNIC together with a PCIe switch. We have confirmed that all our results hold on Bluefield-1 [55]. Moreover, the next generation of Bluefield (Bluefield-3) still follows the same architecture, except for using faster RNIC (400 Gbps ConnectX-7), PCIe (5.0), and SoC (ARMv8.2+ A78). Even though other SmartNICs may have different parameters than Bluefield-2, our methodology, analysis tools (open-sourced), and performance models (e.g., Table 4) also apply to them.

Furthermore, DPDK [1] is another popular communication primitive over SmartNIC. From a NIC’s perspective, DPDK is similar to SEND/RECV over UD. Therefore, we believe that most of our findings are still applicable to DPDK as well. Unfortunately, we do not have an Ethernet-based testbed to confirm this further.

**Suggestions for hardware vendors.** Our study has uncovered several anomalies that can be mitigated through hardware improvements, which we suggest vendors consider. For example, current host to SoC DMA must offload to SoC for execution [56], while supporting CXL [15] can utilize the more powerful host CPU DMA engine for it. However, doing so in a programmer-friendly way [21] will require strong cooperation between the SoC OS and host OS. To the best of our knowledge, no SmartNIC supports CXL yet. Moreover, supporting CCI [5] can mitigate the performance degradation problem described in Advice #1. Furthermore, aligning the SoC PCIe MTU with the host is likely to improve PCIe performance when transferring large payloads. Finally, we encourage vendors to disclose more hardware details of SmartNICs to help explain and confirm the findings of our study.

## 7 Other Related Work

**SmartNIC offloading.** Offloading computation to SmartNICs has attracted significant attention in academia and industry. The offloaded tasks include network functions [59, 34, 19], microservices [14, 39], and others [33, 36, 22, 35, 61, 74, 65]. We share the same vision—improving the performance of distributed systems by offloading computation and communication to SmartNICs, but further exploit the multiple communication paths of SmartNICs. In addition, most prior work has focused on leveraging a single path of on-path SmartNICs, so our work can inspire future research on multi-path offloading for on-path SmartNICs.

**RDMA offloading.** Before the emergence of SmartNICs, many distributed systems offloaded remote memory accesses to one-sided RDMA primitives [75, 64, 17, 63, 46, 50, 13, 76, 82, 79, 84, 40, 85, 86]. However, prior work has observed the poor semantics of one-sided RDMA and has therefore leveraged advanced RDMA features (e.g., WAIT [60, 31], DCT [77, 78]) or introduced new RDMA primitives [65, 10]. These efforts are orthogonal to our work and could also benefit from our findings when using SmartNICs in the future.

## 8 Conclusion

Designing high-performance distributed systems with SmartNICs requires an in-depth understanding of low-level hardware details. This paper presents a comprehensive study of off-path SmartNIC. Unlike prior work, we explore how the SmartNIC architecture and the heterogeneity of its computation units can impact communication performance related to its components. We further propose the first optimization guideline for designers to smartly exploit multiple communication paths of SmartNICs for distributed systems, and demonstrate our guideline by improving two distributed systems. In general, our study can help system designers develop a better understanding of SmartNICs before applying them in high-performance distributed systems.

## Acknowledgment

We sincerely thank our shepherd, Rachit Agarwal, and the anonymous reviewers for their comments and suggestions to improve the paper. We also thank Zhuobin Huang for discussing DPDK on SmartNIC, Jun Lu and Dong Du for providing the Bluefield-1 hardware, Dingji Li, Erhu Feng, Jinyu Gu, and Fangming Lu for their valuable feedback on earlier versions of the paper. This work was supported in part by the National Key Research & Development Program of China (No. 2022YFB4500700), the Fundamental Research Funds for the Central Universities, the National Natural Science Foundation of China (No. 62202291, 62272291, 61925206), as well as research grants from Huawei Technologies and Shanghai AI Laboratory. Corresponding author: Rong Chen ([rongchen@sjtu.edu.cn](mailto:rongchen@sjtu.edu.cn)).



## References

- [1] Data plane development kit. <https://www.dpdk.org/>, 2023.
- [2] AMARO, E., LUO, Z., OUSTERHOUT, A., KRISHNAMURTHY, A., PANDA, A., RATNASAMY, S., AND SHENKER, S. Remote memory calls. In *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020* (2020), B. Y. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, Eds., ACM, pp. 38–44.
- [3] ANDERSON, T. E., CANINI, M., KIM, J., KOSTIC, D., KWON, Y., PETER, S., REDA, W., SCHUH, H. N., AND WITCHEL, E. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1011–1027.
- [4] ANDERSON, T. E., CANINI, M., KIM, J., KOSTIC, D., KWON, Y., PETER, S., REDA, W., SCHUH, H. N., AND WITCHEL, E. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1011–1027.
- [5] ARM. Corelink CCI-550. <https://developer.arm.com/Processors/CoreLink%20CCI-550>, 2022.
- [6] ASSOCIATION., I. T. Infiniband architecture specification. <https://cw.infinibandta.org/document/dl/7859>, 2022.
- [7] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008* (2008), A. Moshovos, D. Tarditi, and K. Olukotun, Eds., ACM, pp. 72–81.
- [8] BROADCOM. Bcm57504 - 100gbe. <https://en.broadcom.com/products/ethernet-connectivity/network-adapters/bcm57504-100g-ic>, 2022.
- [9] BROADCOM. Product Brief: Stingray PS225. <https://docs.broadcom.com/doc/PS225-PB>, 2022.
- [10] BURKE, M., DHARANIPRAGADA, S., JOYNER, S., SZEKERES, A., NELSON, J., ZHANG, I., AND PORTS, D. R. K. PRISM: rethinking the RDMA interface for distributed systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 228–242.
- [11] CHEN, H., CHEN, R., WEI, X., SHI, J., CHEN, Y., WANG, Z., ZANG, B., AND GUAN, H. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst.* 35, 1 (2017), 3:1–3:37.
- [12] CHEN, Y., LU, Y., AND SHU, J. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 19:1–19:14.
- [13] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 26:1–26:17.
- [14] CHOI, S., SHAHBAZ, M., PRABHAKAR, B., AND ROSENBLUM, M.  $\lambda$ -nic: Interactive serverless compute on programmable smartnics. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020* (2020), IEEE, pp. 67–77.
- [15] CONSORTIUM, C. Cxl specification. <https://www.computeexpresslink.org/download-the-specification>, 2022.
- [16] COOPER, B. F. YCSB Core Workloads. <https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads>, 2021.
- [17] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.
- [18] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 54–70.
- [19] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A. M., CHUNG, E. S., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. G. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018), S. Banerjee and S. Seshan, Eds., USENIX Association, pp. 51–66.
- [20] GAO, Y., LI, Q., TANG, L., XI, Y., ZHANG, P., PENG, W., LI, B., WU, Y., LIU, S., YAN, L., FENG, F., ZHUANG, Y., LIU, F., LIU, P., LIU, X., WU, Z., WU, J., CAO, Z., TIAN, C., WU, J., ZHU, J., WANG, H., CAI, D., AND WU, J. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 519–533.
- [21] GOUK, D., LEE, S., KWON, M., AND JUNG, M. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 287–294.



- [22] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. Smartnic performance isolation with fairnic. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (2020), H. Schulzrinne and V. Misra, Eds., ACM, pp. 681–693.
- [23] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.
- [24] HARCHOL-BALTER, M. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.
- [25] HASSAN, H., VIJAYKUMAR, N., KHAN, S. M., GHOSE, S., CHANG, K. K., PEKHIMENKO, G., LEE, D., ERGIN, O., AND MUTLU, O. Softmc: A flexible and practical open-source infrastructure for enabling experimental DRAM studies. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017* (2017), IEEE Computer Society, pp. 241–252.
- [26] INTEL. Intel® data direct i/o technology. <https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html>, 2022.
- [27] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014* (2014), F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds., ACM, pp. 295–306.
- [28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.
- [29] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.
- [30] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Data-center rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 1–16.
- [31] KIM, D., MEMARIPOUR, A. S., BADAM, A., ZHU, Y., LIU, H. H., PADHYE, J., RAINDEL, S., SWANSON, S., SEKAR, V., AND SESHAN, S. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (2018), S. Gorinsky and J. Tapolcai, Eds., ACM, pp. 297–312.
- [32] KIM, J., JANG, I., REDA, W., IM, J., CANINI, M., KOSTIC, D., KWON, Y., PETER, S., AND WITCHEL, E. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 756–771.
- [33] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017* (2017), ACM, pp. 137–152.
- [34] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., AND CHENG, P. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 1–14.
- [35] LI, J., LU, Y., WANG, Q., LIN, J., YANG, Z., AND SHU, J. Alnico: Smartnic-accelerated contention-aware request scheduling for transaction processing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), J. Schindler and N. Zilberman, Eds., USENIX Association, pp. 951–966.
- [36] LIN, J., PATEL, K., STEPHENS, B. E., SIVARAMAN, A., AND AKELLA, A. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 243–259.
- [37] LIU, J., MALTZAHN, C., ULMER, C. D., AND CURRY, M. L. Performance characteristics of the bluefield-2 smartnic. *CoRR abs/2105.06619* (2021).
- [38] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (2019), J. Wu and W. Hall, Eds., ACM, pp. 318–333.
- [39] LIU, M., PETER, S., KRISHNAMURTHY, A., AND PHOTHILIMTHANA, P. M. E3: energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafir, Eds., USENIX Association, pp. 363–378.
- [40] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 773–785.
- [41] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012* (2012), P. Felber, F. Bellosa, and H. Bos, Eds., ACM, pp. 183–196.

- [42] MARVELL. Marvell liquidio iii. <https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf>, 2022.
- [43] MARVELL. Marvell® octeon 10 dpu platform. <https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf>, 2023.
- [44] MELLANOX. ConnectX-7 product brief. <https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf>, 2022.
- [45] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 451–464.
- [46] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 451–464.
- [47] MONGA, S. K., KASHYAP, S., AND MIN, C. Birds of a feather flock together: Scaling RDMA rpcs with flock. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 212–227.
- [48] NETRONOME. Netronome agilio. <https://www.netronome.com/products/smartnic/overview/>, 2022.
- [49] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (2018), S. Gorinsky and J. Tapolcai, Eds., ACM, pp. 327–341.
- [50] NOVAKOVIC, S., SHAN, Y., KOLLI, A., CUI, M., ZHANG, Y., ERAN, H., PISMENNY, B., LISS, L., WEI, M., TSAFRIR, D., AND AGUILERA, M. K. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019* (2019), M. Hershcovitch, A. Goel, and A. Morrison, Eds., ACM, pp. 97–108.
- [51] NVIDIA. Innova-2 flex. <https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/>, 2022.
- [52] NVIDIA. Nvidia bluefield dpu-2. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf>, 2022.
- [53] NVIDIA. Nvidia bluefield dpu-3. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf>, 2022.
- [54] NVIDIA. Performance Monitoring Counters, BlueField SW Manual v2.4.0.11082. <https://docs.nvidia.com/networking/display/BlueFieldSWv24011082/Performance+Monitoring+Counters>, 2022.
- [55] NVIDIA. Nvidia bluefield dpu-1. <https://docs.nvidia.com/networking/display/BFVPIDPU/Specifications>, 2023.
- [56] NVIDIA. Nvidia doca dma programming guide. <https://docs.nvidia.com/doca/sdk/dma-programming-guide/index.html>, 2023.
- [57] NVIDIA. Nvidia doca software framework. <https://developer.nvidia.com/networking/doca>, 2023.
- [58] OLUMIDE OLUSANYA AND MUNIRA HUSSAIN. Need for Speed: Comparing FDR and EDR InfiniBand (Part 1). [https://dl.dell.com/manuals/all-products/esuprt\\_software/esuprt\\_it\\_ops\\_datcentr\\_mgmt/high-computing-solution-resources\\_white-papers77\\_en-us.pdf](https://dl.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers77_en-us.pdf), 2022.
- [59] PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODÍK, R., AND ANDERSON, T. E. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 663–679.
- [60] REDA, W., CANINI, M., KOSTIC, D., AND PETER, S. RDMA is turing complete, we just did not know it yet! *CoRR abs/2103.13351* (2021).
- [61] SCHUH, H. N., LIANG, W., LIU, M., NELSON, J., AND KRISHNAMURTHY, A. Xenic: Smartnic-accelerated distributed transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 740–755.
- [62] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIC, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 433–448.
- [63] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIC, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 433–448.

- [64] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 317–332.
- [65] SIDLER, D., WANG, Z., CHIOSA, M., KULKARNI, A., AND ALONSO, G. Strom: smart remote memory. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 29:1–29:16.
- [66] TANG, C., WANG, Y., DONG, Z., HU, G., WANG, Z., WANG, M., AND CHEN, H. Xindex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2020), PPOPP '20, Association for Computing Machinery, p. 308–320.
- [67] THOMAS, S., VOELKER, G. M., AND PORTER, G. Cachecloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2018, Boston, MA, USA, July 9, 2018* (2018), G. Ananthanarayanan and I. Gupta, Eds., USENIX Association.
- [68] THOSTRUP, L., FAILING, D., ZIEGLER, T., AND BINNIG, C. A dbms-centric evaluation of bluefield dpus on fast networks. In *13th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures* (2022).
- [69] TIMOTHY PRICKETT MORGAN. Pushing PCI-express fabrics up to the next level. <https://www.nextplatform.com/2020/03/27/pushing-pci-express-fabrics-up-to-the-next-level/>, 2022.
- [70] TSAI, S., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 33–48.
- [71] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.
- [72] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (2004), E. A. Brewer and P. Chen, Eds., USENIX Association, pp. 91–104.
- [73] WANG, X., KOTRA, J. B., AND JIAN, X. Eager memory cryptography in caches. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022* (2022), IEEE, pp. 693–709.
- [74] WANG, Z., HUANG, H., ZHANG, J., WU, F., AND ALONSO, G. Fpganic: An fpga-based versatile 100gb smartnic for gpus. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), J. Schindler and N. Zilberman, Eds., USENIX Association, pp. 967–986.
- [75] WEI, X., CHEN, R., AND CHEN, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.
- [76] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 233–251.
- [77] WEI, X., LU, F., CHEN, R., AND CHEN, H. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 121–136.
- [78] WEI, X., LU, F., WANG, T., GU, J., YANG, Y., CHEN, R., AND CHEN, H. No provisioned concurrency: Fast rdma-coded remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association.
- [79] WEI, X., SHEN, S., CHEN, R., AND CHEN, H. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, 2017), USENIX ATC '17, USENIX Association, pp. 335–347.
- [80] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 87–104.
- [81] WEI, X., XIE, X., CHEN, R., CHEN, H., AND ZANG, B. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 523–536.
- [82] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving Graph Traversal with Split Live Migration. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), pp. 723–738.
- [83] XILLYBUS. Down to the tlp: How pci express devices talk. <http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1>, 2022.
- [84] ZAMANIAN, E., BINNIG, C., KRASKA, T., AND HARRIS, T. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.* 10, 6 (2017), 685–696.
- [85] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 614–630.
- [86] ZUO, P., SUN, J., YANG, L., ZHANG, S., AND HUA, Y. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 15–29.





# Ensō: A Streaming Interface for NIC-Application Communication

Hugo Sadok 🐼 Nirav Atre 🐼 Zhipeng Zhao 🇺🇸 Daniel S. Berger 🇺🇸🇨🇦

James C. Hoe 🐼 Aurojit Panda 🇮🇳 Justine Sherry 🐼 Ren Wang 🇮

🐼 Carnegie Mellon University

🇮 Intel

🇺 Microsoft

🇺 New York University

🇺 University of Washington

## Abstract

Today, most communication between the NIC and software involves exchanging fixed-size packet buffers. This packetized interface was designed for an era when NICs implemented few offloads and software implemented the logic for translating between application data and packets. However, both NICs and networked software have evolved: modern NICs implement hardware offloads, e.g., TSO, LRO, and serialization offloads that can more efficiently translate between application data and packets. Furthermore, modern software increasingly batches network I/O to reduce overheads. These changes have led to a mismatch between the packetized interface, which assumes that the NIC and software exchange fixed-size buffers, and the features provided by modern NICs and used by modern software. This incongruence between interface and data adds software complexity and I/O overheads, which in turn limits communication performance.

This paper proposes Ensō, a new streaming NIC-to-software interface designed to better support how NICs and software interact today. At its core, Ensō eschews fixed-size buffers, and instead structures communication as a stream that can be used to send arbitrary data sizes. We show that this change reduces software overheads, reduces PCIe bandwidth requirements, and leads to fewer cache misses. These improvements allow an Ensō-based NIC to saturate a 100 Gbps link with minimum-sized packets (forwarding at 148.8 Mpps) using a single core, improve throughput for high-performance network applications by 1.5–6×, and reduce latency by up to 43%.

## 1 Introduction

Network performance dictates application performance for many of today’s distributed and cloud computing applications [48]. While growing application demands have led to a rapid increase in link speeds from 100 Mbps links [31] in 2003 to 100 Gbps in 2020 [89] and 200 Gbps in 2022 [58], a slowdown in CPU scaling has meant that applications often cannot fully utilize these links. Consequently, recent changes to NICs and networked software have focused on reducing the number of CPU cycles required for communication: NIC offloads allow the NIC to perform common tasks (e.g., segmentation) previously implemented in software; and more efficient network I/O libraries and interfaces, including DPDK and XDP, allow applications to reduce processing in the network stack. We begin with the observation that despite these changes, utilizing 100 Gbps or 400 Gbps links

remains challenging. We demonstrate that this is because of inefficiencies in how software communicates with the NIC. While NICs and the software that communicate with them have themselves changed significantly in the last decade, the NIC-to-software interface has remained unchanged for decades.<sup>1</sup>

Most NICs currently provide an interface where all communication between software and the NIC requires sending (and receiving) a sequence of fixed-size buffers, which we call *packet buffers* in this paper. Packet buffer size is dictated by software, and is usually chosen to be large enough to fit MTU-sized packets, e.g., Linux uses 1536 byte packet buffers (sk\_bufs) and DPDK [19] uses 2kB packet buffers (mbufs) by default. We use the term *packetized NIC interface* to refer to any NIC-to-software interface that uses packet buffers for communication. We observe that two changes in how NICs are used today have led to an impedance mismatch with packetized interfaces.

First, many NIC offloads such as TCP Segmentation Offloading (TSO) [20, 39], Large Receive Offloading (LRO) [14], serialization offloads [44, 71, 86], and transport offloads [3, 14, 27, 77] take inputs (and produce outputs) that can span multiple packets and vary in size. In using these offloads with a packetized interface, software must needlessly split (and recombine) data into multiple packet buffers when communicating with the NIC.

Second, software logic for sending (and receiving) packets uses batches of multiple packets to reduce I/O overheads. In the common case, NICs and software process packets in a batch sequentially. However, packetized interfaces cannot ensure that packets in a batch are in contiguous and sequential memory locations, reducing the effectiveness of several CPU and IO optimizations.

This mismatch between how modern NICs are used and what the packetized interface provides causes three problems that affect application performance:

**Packetized abstraction:** While imposing fixed-size buffers works reasonably well when software *always* needs to exchange MTU-sized packets, it becomes clumsy when used with higher-level abstractions such as application-level messages (e.g., RPCs), bytestreams, or even simpler offloads such as LRO. When using this interface, the NIC (or software) must split messages that are larger than the packet buffer into multiple packet buffers. Applications then need to deal

<sup>1</sup>Osiris [22], published in 1994, describes an interface that is nearly identical to the one adopted by many modern NICs.



with input that is split across multiple packet buffers. Doing so either requires that they first copy data to a separate buffer, or that the application logic itself be designed to deal with packetized data. Indeed, implementing any offload or abstraction that deals with more than a single packet’s worth of data (e.g., transport protocols, such as TCP, that provide a bytestream abstraction) in a NIC that implements the packetized interface requires copying data from packet buffers to a stream. This additional copy can add significant overhead, negating some of the benefits of such offloads [72, 88].

**Poor cache interaction:** Because the packetized interface forces incoming and outgoing data to be scattered across memory, it limits the effectiveness of prefetchers and other CPU optimizations that require predicting the next memory address that software will access—a phenomenon that we refer to as *chaotic memory access*. As we show in §7, chaotic memory accesses can significantly degrade application performance, particularly those that deal with small requests such as object caches [9, 57] and key-value stores [4, 52].

**Metadata overhead:** Since the packetized interface relies on per-packet metadata, it spends a significant portion of the PCIe bandwidth transferring metadata—as much as 39% of the available bandwidth when using small messages. This causes applications that deal with small requests to be bottlenecked by PCIe, which prevents them from scaling beyond a certain number of cores. The use of per-packet metadata also contributes to an increase in the number of memory accesses required for software to send and receive data, further reducing the cycles available for the application. We observed scalability issues due to PCIe bottleneck in our implementation of Google’s Maglev Load Balancer [23].

In this paper, we propose Ensō, a new interface for NIC-application communication that breaks from the lower-level concept of packets. Instead, Ensō provides a streaming abstraction that the NIC and applications can use to communicate arbitrary-sized chunks of data. Doing so not only frees the NIC and application to use arbitrary data formats that are more suitable for the functionality implemented by each one but also moves away from the performance issues present in the packetized interface. Because Ensō makes no assumption about the data format itself, it can be repurposed depending on the application and the offloads enabled on the NIC. For instance, if the NIC is only responsible for multiplexing/demultiplexing, it can use Ensō to deliver raw packets; if the NIC is also aware of application-level messages, it can use Ensō to deliver entire messages and RPCs to the application; and if the NIC implements a transport protocol, such as TCP, it can use Ensō to communicate with the application using bytestreams.

To provide a streaming abstraction, Ensō replaces ring buffers containing *descriptors*, used by the current NIC interface, with a ring buffer containing *data*. The NIC and the software communicate by appending data to these ring

buffers. Ensō treats buffers as *opaque* data, and does not impose any requirements on their content, structure or size, thus allowing them to be used to transfer arbitrary data, whose size can be as large as the ring buffer itself. Ensō also significantly reduces PCIe bandwidth overhead due to metadata, because it is able to aggregate notifications for multiple chunks of data written to the same buffer. Finally, it enables better use of the CPU prefetcher to mask memory latency, thus further improving application performance.

Although the insight behind this design is simple, it is challenging to implement in practice. For example, CPU-NIC synchronization can easily lead to poor cache performance: any approach where the NIC and CPU poll for changes at a particular memory location will lead to frequent cache invalidation. Ensō avoids this obstacle by relying on explicit notifications for CPU-NIC synchronization. Unfortunately, explicit notifications require additional metadata to be sent over the CPU-NIC interconnect, which can negate any benefits for interconnect bandwidth utilization. Ensō mitigates this overhead by sending notifications *reactively*. We discuss our synchronization strategy in detail, as well as other challenges to the Ensō design in §4.

To understand its performance, we fully implement Ensō using an FPGA-based SmartNIC. We describe our hardware and software implementations in §5 and how Ensō can be used depending on the functionality offered by the NIC in §6. In §7 we present our evaluation of Ensō, including its use in four applications: the Maglev load balancer [23], a network telemetry application based on NitroSketch [54], the MICA key-value store [52], and a log monitor inspired by AWS CloudWatch Logs [6]. We also implemented a software packet generator that we use in most of the experiments.<sup>2</sup> We observe speedups of up to 6× relative to a DPDK implementation for Maglev, and up to 1.47× for MICA with no hardware offloads.

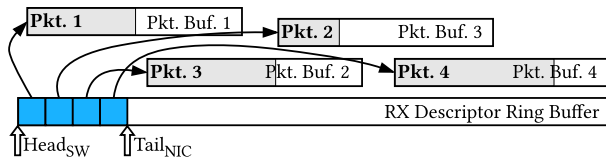
Finally, while Ensō is optimized for applications that process data in order, we show that Ensō also outperforms the existing packetized interface when used by applications that process packets out of order (e.g., virtual switches), despite requiring an additional memory copy (§7.2.2).

Ensō is fully open source, with our hardware and software implementations available at <https://enso.cs.cmu.edu/>.

## 2 Background and Motivation

The way software (either the kernel or applications using a kernel-bypass API) and the NIC exchange data is defined by the interface that the NIC hardware exposes. Today, most NICs expose a *packetized* NIC interface. This includes NICs from several companies including Amazon [2], Broadcom [12], Intel [39], Marvell [56], and others. Indeed, prior

<sup>2</sup>Developing this software packet generator was a necessary first step in evaluating Ensō because no existing software packet generators could scale to the link rates we needed to stress test Ensō!



**Figure 1:** Data structures used to receive packets in a packetized NIC interface. Each packet is placed in a separate buffer that can be arranged arbitrarily in memory.

work [68] found that of the 44 NIC drivers included in DPDK, 40 use this interface. Due to its ubiquity, the packetized NIC interface has dictated the API provided by nearly all high-performance network libraries, including `io_uring` [15], DPDK [19] and `netmap` [73]. In this section, we describe the packetized NIC interface and highlight some of the issues that it brings to high-performance applications.

## 2.1 Packetized NIC Interface

A core design choice in the packetized NIC interface is to place every packet in a dedicated packet buffer. The NIC and the software communicate by exchanging packet *descriptors*. Descriptors hold metadata, including packet size, what processing the NIC should perform (e.g., update the checksum or segment the packet), a flag bit, and a pointer to a separate *packet buffer* which holds the actual packet data. Most packet processing software pre-allocate a fixed number of buffers for packets; new packets (either generated by an application or incoming from the network) are assigned to the next available buffer in the pool, which may not reside in memory anywhere near the preceding or following packet. Because software does not know the size of incoming packets beforehand, buffers are often sized so that they can accommodate MTU-sized packets (e.g., 1536B in Linux and 2kB in DPDK).

Figure 1 shows an example of a packetized NIC interface being used to receive four packets from a particular hardware queue on the NIC. The NIC queue is associated with a set of NIC registers that can be used to control a receive (RX) descriptor ring buffer and a transmit (TX) descriptor ring buffer. Before being able to receive packets, the software informs the NIC of the addresses of multiple available buffers in its pool by enqueueing descriptors pointing to each one in the RX descriptor ring buffer. The NIC can then use DMA to write the incoming packet data into the next available packet buffer and enqueue updated descriptors containing metadata such as the packet size. Importantly, the NIC also sets a ‘flag’ bit in the descriptor to signal to the software that packets have arrived for this buffer. Observing a notification bit for the descriptor under the head pointer, the software can then increment the head pointer.

A similar process takes place for transmission: the sending software assembles a set of descriptors for packet buffers that are ready to be transmitted and copies the descriptors—but not the packets themselves—into the TX ring buffer; the flag bit in the descriptor is now used to signal that the NIC has

transmitted (rather than received) a packet.

One of the major benefits of dedicating buffers for each packet is that multiplexing/demultiplexing can be done efficiently in software. If the software transmitting packets is the kernel, this might mean associating each descriptor/packet pair with an appropriate socket; if the software in use is a software switch [32, 67] this might mean steering the right packet to an appropriate virtual machine. Either way, the cleverness of the packetized NIC interface in using dedicated packet buffers shines here: rather than copying individual packets in the process of sorting through inbound packets, the switching logic can deliver packet pointers to the appropriate endpoints. These packets can then be processed and freed in arbitrary order.

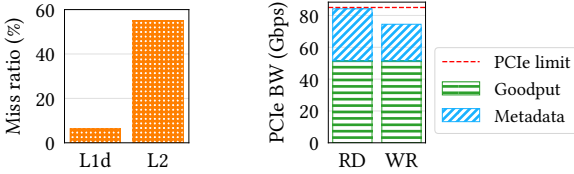
The usage model for a modern high-performance software stack, however, looks very different. Instead of *one* software entity (e.g., kernel, software switch) mediating access to the NIC, there may be many threads or processes with *direct NIC access* (i.e., kernel bypass). High-performance NIC ASICs expose *multiple hardware queues* (as many as thousands [39]) so that each thread or process can transmit and receive data directly to the NIC without coordination between them. The NIC then takes on all of the responsibilities of demultiplexing, using, e.g., RSS [82], Intel’s Flow Director [39], or (for a very rich switching model) Microsoft’s AccelNet [28]. In this setting, the multiplexing/demultiplexing capabilities of the packetized NIC interface offer no additional value.

## 2.2 Issues with a Packetized Interface

While many high-performance applications today gain little from a packetized interface, they still need to pay for the overheads accompanying it. Shoehorning data communication between the NIC and applications into fixed-sized chunks leads to inefficient use of CPU caches and PCIe bandwidth for small requests, as well as additional data copies due to fragmentation for applications that rely on large messages or bytestreams.

In this section, we conduct microbenchmarks that isolate these issues, and in §7, we also show the impact that these issues have on real applications.

**Chaotic Memory Access:** We experiment with a simple DPDK-based ping/pong program (a description of our testbed is in §7) which receives a packet, increments a byte in the packet payload, and re-transmits it. For this program, we observed maximum throughput of 40 Gbps using a 100 Gb NIC (Intel E810) and a single 3.1 GHz CPU core. When we conduct a top-down analysis [43], we see that the application is backend-bound, primarily due to L1 and L2 cache misses. Figure 2a shows around 6% miss ratio for the L1d and a 55% miss ratio for the L2 cache. This high cache miss ratio is a direct consequence of using per-packet buffers in the packetized NIC interface. First, because *packet buffers themselves are scattered in memory, reads and writes to packet data evade*



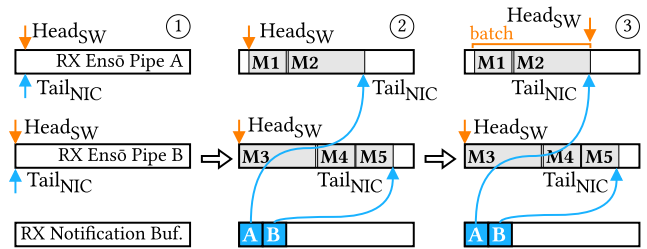
(a) Miss ratio for L1d and L2 caches. (b) PCIe bandwidth utilization. Up to 39% of the read bandwidth is consumed with metadata.

**Figure 2:** PCIe bandwidth and cache misses for an application forwarding small packets with a packetized NIC interface (E810).

any potential benefit from shared cache lines or prefetching.<sup>3</sup> Applications like key-value stores [4, 52] or packet processors [18] exhibit very high *spatio-temporal* locality in their data access: they are designed to run to completion (i.e., they continue working on a packet or batch until the work for that item is completed, leading to repeated accesses to the same data), and they operate over incoming packets or batches in the order in which they arrive (i.e., the current item being processed serves as an excellent predictor of the next one). However, this structure is not realized in the memory layout of packetized buffers, and hence to any cache optimizations, reads and writes appear unpredictable. Second, because every packet is paired with a descriptor, the total amount of memory required to store all of the data required for I/O increases, exacerbating last-level cache contention simply because more data needs to be accessed. Indeed, prior work [55, 81] has repeatedly demonstrated that the size of the working set for packet processing applications often outgrows the amount of cache space dedicated to DDIO [35], negating the benefits of this hardware optimization to bring I/O data directly into the cache. As we discuss in detail in §7.2.3, using a different NIC interface that facilitates sequential memory accesses can drop the miss ratio from 6% to 0.2% for the L1d cache, and from 55% to 9% for the L2 cache.

**Metadata Bandwidth Overhead:** We observe that the packetized NIC interface requires the CPU and the NIC to exchange *both* descriptors and packet buffers. This leads to the second problem with the packetized interface: *up to 39% of the CPU to NIC interconnect bandwidth is spent transferring descriptors* (Figure 2b). While NIC-CPU interconnect line rates are typically higher than network line rates, the gap between them is relatively narrow. This is particularly problematic for small transfers as the PCIe theoretical limit drops to only 85 Gbps with 64-byte transfers [62]. We also expect this gap to remain small in the future as a state-of-the-art next generation server with a 400 Gbps Ethernet connection and 512 Gbps of PCIe 5.0 bandwidth would still bottleneck with 39% of bandwidth wasted on metadata. This observation complements recent studies that also point to the PCIe as a source of congestion for transport protocols [1].

<sup>3</sup>We note here that the aforementioned performance penalty arises in spite of the fact that DPDK performs mbuf-level software prefetching.



**Figure 3:** Steps to receive batches of messages in two Ensō Pipes.

**In summary:** By pairing every packet with a separate descriptor, the packetized NIC interface was well designed for a previous generation of high-throughput networked applications which needed to implement multiplexing in software. However, for today’s high-performance applications, it introduces unnecessary performance overheads.

### 3 Ensō Overview

Ensō is a new streaming interface for NIC-application communication. Ensō’s design has three primary goals: (1) *flexibility*, allowing it to be used for different classes of offloads operating at different network layers and with different data sizes; (2) *low software overhead*, reducing the number of cycles that applications need to spend on communication; and (3) *hardware simplicity*, enabling practical implementations on commodity NICs.

Ensō is designed around the *Ensō Pipe*, a new buffer abstraction that allows applications and the NIC to exchange arbitrary chunks of data as if reading and writing to an unbounded memory buffer. Different from the ring buffers employed by the packetized interface (which hold descriptors to scattered packet buffers), an Ensō Pipe is implemented as a *data ring buffer* that contains the actual packet data.

**High-level operation:** In Figure 3 we show how an application, with two Ensō Pipes, receives messages. Initially, the Ensō Pipes are empty, and the HeadSW and TailNIC point to the same location in the buffer ①. When the NIC receives messages, it uses DMA to enqueue them in contiguous memory owned by the Ensō Pipes ②. In the figure, the NIC enqueues two messages in Ensō Pipe A’s memory, and three in Ensō Pipe B’s memory. The NIC informs the software about this by also enqueueing two notifications (one for each Ensō Pipe) in the notification buffer. The software uses these notifications to advance TailNIC and process the messages. Once the messages have been processed, the software writes to a Memory-Mapped I/O (MMIO) register (advancing HeadSW) to notify the NIC—allowing the memory to be reused by later messages ③. Sending messages is symmetric, except for the last step: the NIC notifies the software that messages have been transmitted by overwriting the notification that the CPU used to inform the NIC that a message was available to be transmitted.



**Ensō Pipe’s flexibility:** Although Figure 3 shows the steps to send *messages*, because Ensō Pipes are opaque, they can be used to transmit arbitrary chunks of data. These can be raw packets, messages composed of multiple MTU-sized packets, or even an unbounded bytestream. The format of the data is dictated by the application and the offloads running on the NIC. Moreover, Ensō Pipes’ opaqueness means that they can be mapped to any pinned memory within the application’s memory space. Thus, by mapping both the RX and TX Ensō Pipes to the same region, network functions and other forwarding applications can avoid copying packets. In our evaluation (§7) we use this approach when implementing Maglev and a Network Telemetry application.

**Performance advantages of an Ensō Pipe:** The fact that data can be placed back-to-back inside an Ensō Pipe addresses both of the performance challenges we listed previously: First, Ensō Pipes allow applications to read and write I/O data *sequentially*, thus avoiding chaotic memory accesses. Second, as shown in Figure 3, inlining data in an Ensō Pipe removes the need for per-packet descriptors, thus reducing the amount of metadata exchanged over the PCIe bus, and reducing cycles spent managing (i.e., allocating and freeing) packet buffers.

**Challenges:** Although implementing a ring buffer for data transfer is, on its own, a simple idea, coordinating the notifications between the CPU and the NIC to update head and tail pointers turns out to be challenging.

*Efficient coordination:* The packetized interface coordinates incoming and outgoing packets by ‘piggybacking’ notifications in the descriptor queue itself. Each descriptor includes a ‘flag bit’ that can be used to signal when the descriptor is valid. Software polls the next descriptor’s flag bit to check if a new packet arrived. We cannot use the same strategy for Ensō Pipes as they do not assume a format for the data in the buffer, and hence cannot embed control signals in it.

In §4.1, we discuss how naïve approaches to notification can stress worst-case performance of MMIO and DMA. In particular, concurrent accesses to the same memory address can create cache contention between the CPU and the NIC. Ensō uses dedicated *notification buffers* to synchronize updates to head and tail pointers; when combined with batching and multiqueue processing, the notification buffer approach reduces the threat of cache contention.

*Notification pacing:* Ensō Pipes are designed so that notifications for multiple packets can be combined, reducing the amount of metadata transferred between the CPU and the NIC. However, it is still important to decide *when* to send notifications: when sent too frequently they waste PCIe bandwidth and add software overheads, but if sent too infrequently the core might be idle waiting for notification, thus reducing throughput. Ensō includes two mechanisms, *reactive notifications* and *notification prefetching* (§4.2), that

control when notifications are sent. These mechanisms are naturally adaptive, i.e., they minimize the number of notifications sent without limiting throughput, and can be implemented without adding hardware complexity.

*Low hardware complexity and state:* Because the design of Ensō involves both hardware and software, we must be careful to not pay for software simplicity with hardware complexity. Ensō favors coordination mechanisms that require little NIC state. We aim for a design that is simple and easily parallelized. We present Ensō’s hardware design in §5.

**Target applications:** Ensō implements a streaming interface that is optimized for cases where software processes received data in order. Our evaluation (§7) shows that this covers a wide range of network-intensive applications.

One might expect that the resulting design is ill-suited for applications that need to multiplex and demultiplex packets (e.g., virtual switches like Open vSwitch [67] and BESS [32]), as such applications require additional copies with Ensō.<sup>4</sup> However, perhaps surprisingly, Ensō outperforms the packetized interface *even when it requires such additional copies*. As we show in §7.2.2, when comparing the performance of an application that uses Ensō and copies each packet, to a similar DPDK-based application that does not copy packets, using a CAIDA trace [13] (average packet size of 462 B), we find that Ensō’s throughput is still 28% higher than DPDK’s (92.6 Gbps vs. 72.6 Gbps). We discuss how Ensō can be used depending on the applications and the functionality offered by the NIC in §6.

## 4 Efficient Notifications

The key challenges in Ensō arise from efficiently coordinating Ensō Pipes between the CPU and the NIC. In this section, we describe how Ensō efficiently coordinates pipes using notifications (§4.1) and how it paces such notifications (§4.2).

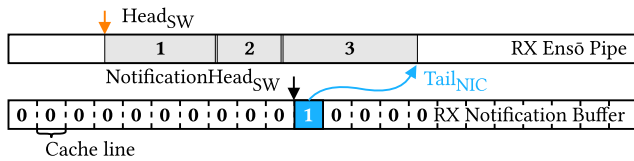
### 4.1 Efficient Ensō Pipe Coordination

Recall from Figure 3 that the software and the NIC coordinate access to RX Ensō Pipes using  $\text{Head}_{\text{SW}}$  and  $\text{Tail}_{\text{NIC}}$  and to TX Ensō Pipes using  $\text{Head}_{\text{NIC}}$  and  $\text{Tail}_{\text{SW}}$ . How should software and the NIC communicate pointer updates?

In the descriptor ring buffers employed by the packetized NIC interface, software communicates pointer updates to the NIC using MMIO writes, and the NIC communicates pointer updates via inline signaling in the descriptor buffer itself [25, 39], avoiding the overheads of MMIO reads. Because the descriptor’s format is defined by the NIC, the NIC

<sup>4</sup>Note that this overhead only affects applications that multiplex/demultiplex packets, and does not apply to software, e.g., TCP stacks, that processes packet data but might need to reorder packets. This is because reordering packet data (rather than whole packets) requires a memory copy when using either interface.





**Figure 4:** Notification buffer being used to notify the arrival of three chunks of data in an RX Ensō Pipe. Notifications contain the latest Tail<sub>NIC</sub> for a given Ensō Pipe as well as a flag signal.

can dedicate a ‘flag signal’ in every descriptor to signal that the descriptor is valid. Software can then poll the next descriptor until its flag becomes valid. This way, there is no need for the NIC to explicitly tell software of pointer updates. Unfortunately, we cannot use this approach for Ensō. While Ensō Pipes can still use MMIO writes for pointer updates from software, we cannot embed inline signaling in the Ensō Pipe itself since we do not impose any structure on the data.

We considered several design options to communicate pointer updates. We focus on one illuminating rejected design here: sharing an address in main memory between the NIC and software. With each Ensō Pipe, we might have a dedicated address in memory where the NIC writes the latest Tail<sub>NIC</sub>. The software can then poll this address to determine the latest value. Unfortunately, this approach leads to contention because every time the NIC writes to memory, the cache entry on the CPU is invalidated. If software continues to poll the same cache line, the resulting contention reduces throughput by orders of magnitude: we measured a throughput below 5 Gbps when using small transfers with this approach. We discuss other rejected approaches in Appendix A.

#### 4.1.1 Notification Buffer

Ensō uses a *notification buffer* to communicate pointer updates. Although the structure of the notification buffer on its own does not solve the cache contention challenge, when combined with *batched notifications* and when it is used to aggregate notification updates to/from *multiple* Ensō Pipes, this approach prevents the CPU from busy waiting on the shared cache line and hence avoids contention-induced slowdowns.

Figure 4 shows an RX Ensō Pipe with its corresponding notification buffer. It shows how a single notification indicates the presence of multiple sequential chunks of data at the same time; we discuss how notifications are ‘batched’ or coalesced in §4.2. Notifications contain the latest Tail<sub>NIC</sub> for a given RX Ensō Pipe as well as a flag signal that software can use to check if the next notification is ready to be consumed. As is done typically with descriptor ring buffers, software advances the NotificationHead<sub>SW</sub> using an MMIO write after consuming a notification. Like an RX Ensō Pipe, a TX Ensō Pipe also uses a separate notification buffer to synchronize pointer updates. But software enqueues new notifications when it wants to transmit a chunk of data and the NIC overwrites the transmission notification with a completion noti-

fication once it is done transmitting the corresponding batch. Completions flip a flag signal, so that software can check if the following cache line corresponds to a completion or a pending TX notification.

#### 4.1.2 Multiplexing and Scaling

**Within a single thread:** To let a single thread efficiently access multiple Ensō Pipes, we associate multiple Ensō Pipes with the same notification buffer. To accomplish this, notifications include an *Ensō Pipe ID* alongside the Tail<sub>NIC</sub> and the flag signal that we discussed before. As a result, software can probe a *single* notification buffer to retrieve updates from *multiple* Ensō Pipes. This avoids the known scalability issues from needing to poll multiple queues [59, 76].

**Among multiple threads:** To let multiple threads send and receive data independently, Ensō supports multiple notification buffers. Each thread can use a *dedicated* notification buffer, avoiding costly synchronization primitives. When setting up a new Ensō Pipe, software tells the NIC which notification buffer is associated with it. Therefore, the NIC knows to which notification buffer to send a notification.

**Among multiple applications:** In addition to using independent notification buffers, Ensō ensures that applications only have access to their own subset of Ensō Pipes and notification buffers. Each queue’s MMIO pointer register pair is kept in its own dedicated page-aligned block of memory [22]. This lets the kernel map the pointer registers at a per-queue granularity to the address space of the application that requested it.

#### 4.1.3 Notifications: Contention and Overhead

Allowing multiple Ensō Pipes to share the same notification queue (§4.1.2), and having notifications arrive only for larger batches of data (§4.2) naturally prevents contention by keeping the NIC ‘ahead’ of the CPU in updating the notification buffer, and also reduces the PCIe overhead of communicating these notifications. As the CPU reads in data for one Ensō Pipe, the NIC is writing new entries for subsequent Ensō Pipes. Because the CPU is processing larger batches of data, it is busier for longer before it needs to check the notification buffer. Hence, as line rates go up, the two are unlikely to be accessing the same cache line simultaneously.

### 4.2 Pacing Notifications

As mentioned above, Ensō batches notifications aiming to reduce metadata bandwidth consumption and to keep CPUs busy processing data, rather than waiting for notifications. Using the wrong batch size, however, is problematic: a system that uses batch sizes that are too small would unnecessarily transmit extra metadata, and a system that uses batch

```

func onPktArrival()      func onRxUpdate(HeadSW)
if status = 0 then      if HeadSW = TailNIC then
    notify(TailNIC)      status ← 0
    status ← 1           else
                        notify(TailNIC)

```

**Figure 5:** Reactive notification mechanism for an Ensō Pipe. It only sends notifications when new data arrives and status = 0, or if software updates Head<sub>SW</sub> and it is different from Tail<sub>NIC</sub>.

sizes that are too large might unnecessarily inflate latency. Ideally, the NIC could keep track of how frequently software is consuming notifications and try to send a notification right before software needs the next one, with all the data that have arrived since the last software read. Ensō approximates this ideal approach but without the impossibility of perfectly predicting when the next read is coming.

Instead of trying to predict when to send the next notification, Ensō lets software dictate the pace of notifications by sending notifications *reactively*. It leverages the fact that the NIC is already aware of when the software is consuming data, as the NIC is notified whenever software updates Head<sub>SW</sub> through MMIO writes. Therefore, the NIC can send notifications in response to these updates. Specifically, we allow the actual NIC tail pointer (Tail<sub>NIC</sub>) to diverge temporarily from what is observed from software via the notification buffer. Ensō suppresses updates to Tail<sub>NIC</sub> until the NIC sees an update to Head<sub>SW</sub>. Our implementation uses a single status bit for every Ensō Pipe, initialized to ‘0’, which indicates if the buffer has data.

**Operation:** Figure 5 summarizes the reactive notification mechanism. Whenever data arrive at an RX Ensō Pipe (onPktArrival), we check the status bit, only sending a notification if status = 0 (indicating that the buffer is empty). We also potentially send notifications whenever software updates Head<sub>SW</sub> (onRxUpdate). If the new Head<sub>SW</sub> is the same as Tail<sub>NIC</sub>, it means that the buffer is now empty and we can reset the status back to ‘0’ without sending a new notification. Otherwise, it indicates that software is unaware of some of the latest data in the buffer, which triggers a new notification.

**Discussion:** Reactive notifications cause the notification rate to naturally adapt to the rate at which software is consuming data, as well as how fast incoming data is arriving. When software is slow to consume data from a particular Ensō Pipe, bytes accumulate and the NIC sends fewer notifications for that Ensō Pipe. When software is fast to consume data, it advances the Head<sub>SW</sub> pointer more often, causing the NIC to send frequent notifications.

Reactive notifications ensure that every piece of data is notified, but as we will see in §7.2.5, they can impose a small latency overhead. This overhead occurs if packet arrivals are known to the NIC but have not yet been communicated in the notification buffer. In this case, when Head<sub>SW</sub> reaches Tail<sub>NIC</sub>, software has to wait for a PCIe RTT before it is notified of the waiting packets. While an extra PCIe RTT

is unlikely to be an issue for Internet-facing applications, it might be an issue for some latency-sensitive applications [7].

**Notification prefetching:** To improve latency for latency-sensitive applications, Ensō also implements a notification prefetching mechanism. Notification prefetching allows software to explicitly request a new notification to the NIC. Applications can use notification prefetching either explicitly, by calling a function to prefetch notifications for a given Ensō Pipe, or transparently, by compiling the Ensō library in low-latency mode. When compiled in low-latency mode, the library always prefetches notifications for the next Ensō Pipe before returning data for the current one. We evaluate the impact of using notification prefetching in §7.2.5.

## 5 Ensō Implementation

Our Ensō implementation consists of three pieces: a userspace library, a kernel module, and a NIC hardware that implements Ensō. We implement the library and kernel module on Linux in about 9k lines of C and C++17. Our hardware implementation uses about 10k lines of SystemVerilog, excluding tests and auto-generated IPs. Our hardware implementation targets an FPGA SmartNIC but the same design could also be implemented in an ASIC.

### 5.1 Software Implementation

Applications use a library call to request notification buffers and Ensō Pipes. Typically, applications will request a notification buffer for each thread but may use multiple Ensō Pipes per thread, depending on their needs. The library sends requests for new Ensō Pipes and notification buffers to the kernel, which checks permissions, allocates them on the NIC, and then maps them into application space. Ensō Pipes are typically allocated in pairs of RX and TX Ensō Pipe but may also be allocated as unified RX/TX Ensō Pipes. Unified RX/TX Ensō Pipes map the RX and TX buffer to the same memory region, which is useful for applications that modify data in place and send them back, e.g., network functions. In contrast, separate Ensō Pipes map the RX and TX buffers to the different memory regions and are useful for typical request-response applications.

To ensure a consistent abstraction of unbounded Ensō Pipes we must also deal with corner cases that arise when data wrap around the buffer limit. To prevent breaking received data that wrap around, we map the same Ensō Pipe’s physical address twice in the application’s virtual memory address space. This means that, to the application, the buffer appears to be twice its actual size, with the second half always mirroring the first one. The application can then consume up to the full size of the buffer of data at once, regardless of where in the buffer the current Head<sub>SW</sub> is. To transmit data that wrap around the buffer limit, the library checks if

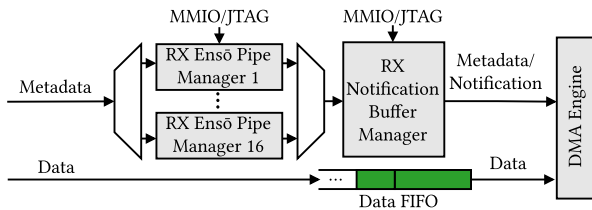


Figure 6: NIC RX Datapath.

the transfer goes around the Ensō Pipe boundary and automatically partitions it into several transfers, each of which is smaller than the overall buffer size.

## 5.2 Hardware Implementation

We now describe key RX and TX hardware modules.

### 5.2.1 RX Datapath

Figure 6 illustrates the RX datapath. It receives as input data and metadata, which includes the Ensō Pipe ID and the size of the corresponding data that is being enqueued. Metadata is handled separately from the data, which allows for smaller queues between modules. The RX datapath is composed of the following modules:

**RX Ensō Pipe Managers:** The Ensō Pipe managers are responsible for keeping Ensō Pipe state such as the buffer’s physical address,  $Head_{SW}$ ,  $Tail_{NIC}$ , notification buffer ID, and notification status bit. When metadata arrive for Ensō Pipe  $i$ , the manager checks for sufficient space in  $i$  and advances  $i$ ’s  $Tail_{NIC}$ . The manager also determines whether to trigger a notification according to the reactive notification strategy (§4.2). To trigger a notification, the manager includes the notification buffer ID and sets a bit in the metadata that is sent to the Notification Buffer Manager.

We use multiple Ensō Pipe Managers for two reasons. First, it enables flow-level parallelism. Each manager requires two cycles to process each metadata, achieving a request rate of 125 Mpps at a 250 MHz clock. To achieve 100 Gbps line rate (148.8 Mpps) we thus need at least two managers. Second, multiple managers enable scaling to high Ensō Pipe counts. We split the state for different Ensō Pipes among different managers, allowing the logic to be closer to the memory that it needs to access. We configure the number of Ensō Pipe Managers at synthesis time with a default of 16, which allows the design to meet timing for up to 16k Ensō Pipes.

**RX Notification Buffer Manager:** This module issues notifications when needed. It spends one cycle for every request and an extra cycle for those that require a notification. If we can suppress notifications for at least 20% of requests, we only need a single notification manager at a 250 MHz clock. A single notification manager is also sufficient since we use fewer notification buffers than Ensō Pipes, e.g., one notification buffer per CPU core. Our implementation defaults to 128 notification buffers but also meets timing with 1024.

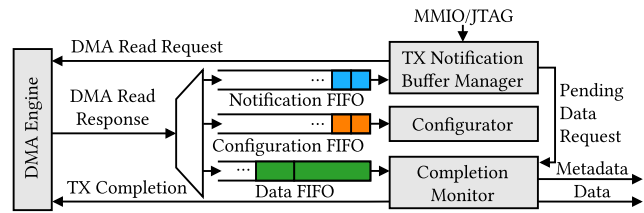


Figure 7: NIC TX Datapath.

**DMA Engine:** This module uses DMA to write data and notifications to the correct address in host memory based on the metadata computed by the upstream modules.

### 5.2.2 TX Datapath and Configuration Path

The TX datapath (Figure 7) is composed of:

**TX Notification Buffer Manager:** This module keeps state for all TX notification buffers. When software enqueues a new TX notification and advances  $NotificationTail_{SW}$  using an MMIO write, the manager requests a DMA read to fetch the notification from memory. The read request is sent to the DMA Engine, which enqueues the DMA read response to the Notification FIFO. The manager can then consume the notifications, allowing it to request DMA reads for the actual data inside an Ensō Pipe. It also sends information about each data request to the Completion Monitor module.

**Completion Monitor:** When the data requested by the TX Notification Buffer Manager arrive, the DMA Engine enqueues them to the Data FIFO. The Completion Monitor consumes the data and keeps track of the number of bytes pending in each request using information that it received from the TX Notification Buffer Manager. When the request completes, the Completion Monitor sends a TX completion notification to the DMA Engine that writes it to host memory. It can then output the data and the metadata, containing the Ensō Pipe ID and size, to downstream modules on the NIC.

**Configurator:** Configuration notifications are enqueued to the Configuration FIFO instead of the Notification FIFO. These are consumed by the Configurator, which directs the configuration to the appropriate module on the NIC. For instance, when the kernel sets a new Ensō Pipe, it inserts an entry in the NIC Flow Table to direct a set of packets to it.

## 6 Using Ensō

Ensō provides a zero-copy streaming abstraction that the NIC and applications can use to exchange data. Thus far, we have shown how this abstraction can be efficiently *implemented*. We now discuss how Ensō should be *used*.

How one uses Ensō depends on how features are split between hardware and software. We consider three settings: (1) Traditional NICs which implement simple offloads, such as checksum and RSS [82], and rely on software implementation for the rest. These can use Ensō Pipes to deliver raw



packets to/from a network stack implemented in software. (2) NICs that implement transport offloads [10, 14, 77, 78] in hardware. These can deliver application-level messages or reassembled bytestreams through the Ensō Pipes. And finally, (3) NICs that implement application logic [41, 49], which can use Ensō Pipes to exchange application data. We elaborate on each of these settings below.

**Traditional NICs:** For traditional NICs that perform simple offloads such as checksum and segmentation, using Ensō is not significantly different than using a packetized interface. In both cases, the network stack is implemented in software and only needs to be changed to use Ensō Pipes. Ensō is designed to support several Ensō Pipes, and for most applications this change does not induce any additional software overheads. Furthermore, high-performance packet processing applications that process packets in order, as is the case for most network functions [68] and applications that use UDP, can consume and transmit raw packets through an Ensō Pipe without copies.

However, applications such as virtual switches [32, 67], that multiplex and demultiplex packets (but do not perform reassembly or other functions that require re-ordering packets, where both interfaces require copies), need to perform an additional copy when forwarding packets to their destination. As we show in §7.2.2, Ensō’s performance advantages can outweigh the cost of copies even for such applications.

**NICs with transport offload:** NICs that implement message-based transports (e.g., SCTP, Homa [60]) or streaming-based transports (e.g., TCP) may also choose to use Ensō Pipes to deliver messages or reassembled bytestreams directly to the application without copies.

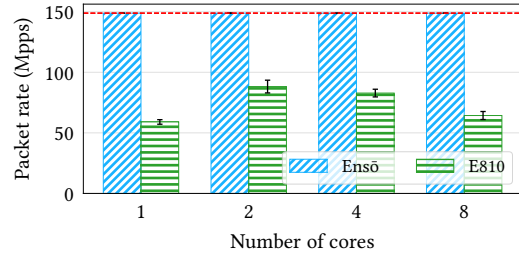
**NICs with application logic:** NICs that implement application-layer protocols or include part of the application logic may use Ensō Pipes to exchange application-level messages with applications. For instance, a NIC that is aware of both TCP and HTTP may deliver incoming HTTP requests back to back to a web server, effectively converting the application to a run-to-completion model.

## 7 Evaluation

We now evaluate our design decisions using microbenchmarks, and then use four real-world applications to show how Ensō improves end-to-end performance.

### 7.1 Setup and Methodology

**Device Under Test (DUT):** We synthesize and run the Ensō NIC on an Intel Stratix 10 MX FPGA NIC [42] with 100 Gb Ethernet and a PCIe 3.0 x16 interface. Most of the NIC design runs at 250 MHz. Our baseline uses an Intel E810 NIC [40] with 100 Gb Ethernet and a PCIe 4.0 x16 interface, and uses



**Figure 8:** Raw packet rate. Ensō is bottlenecked by Ethernet while the E810 does not scale beyond two cores. The dashed line represents the 100 Gb Ethernet limit.

DPDK to minimize software overheads. All our experiments are run on a server with an Intel Core i9-9960X CPU [38] with 16 cores running at 3.1 GHz base frequency, 22 MB of LLC, and PCIe 3.0. We disable dynamic frequency scaling, hyper-threading, power management features (C-states and P-states), and isolate CPU cores from the scheduler.

**Packet generator:** The packet generator machine is equipped with an Intel Core i7-7820X CPU [37] with 8 cores running at 3.6 GHz base frequency, 11 MB of LLC, and PCIe 3.0. It includes another Stratix 10 MX FPGA connected back to back to the E810 and the FPGA on the DUT machine.

We found that existing high-performance packet generators such as DPDK Pktgen [85] and Moongen [24] are unable to keep up with Ensō’s packet rate because their performance is limited by the packetized NIC interface. We thus implement EnsōGen, a packet generator based on Ensō. EnsōGen generates packets from a pcap file, and can send and receive arbitrary-sized packets at 100 Gbps line rate using a *single* CPU core. We describe EnsōGen in more detail in Appendix B. We use EnsōGen in all experiments except for MICA, where we send requests from a MICA client (§7.3.3).

**Methodology:** We measure zero-loss throughput as defined in RFC 2544 [11, 61] with a precision of 0.1 Gbps. We report median throughput and error bars for one standard deviation from ten repetitions. We measure latency by implementing hardware timestamping on the FPGA, which achieves 5 ns precision for packet RTTs. EnsōGen keeps a histogram with the RTT of every received packet, which we use to compute median and 99<sup>th</sup> percentile latencies. PCIe bandwidth measurements use PCM [17] and we obtain other CPU counters using perf [65]. To evaluate MICA, we use the same methodology as the original paper [52] for consistency.

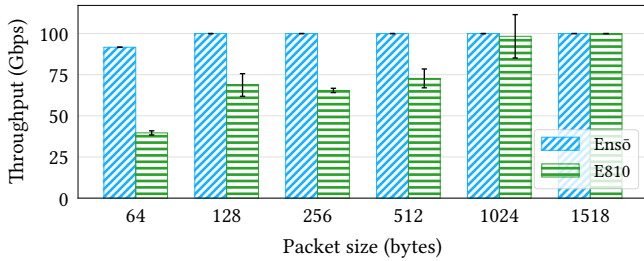
### 7.2 Microbenchmarks

We start by using microbenchmarks to evaluate Ensō’s performance and the design decisions we made.

#### 7.2.1 Packet Rate

We start by measuring how fast Ensō can process packets. We compare the performance of an Ensō-based echo server to that of a DPDK-based echo server. On receiving a packet,





**Figure 9:** Throughput when forwarding packets between two queues. Ensō outperforms zero-copy E810 even when it needs to copy packets.

both versions increment a value in each packet’s payload and then send the packet back out through the same interface. We increment the payload value to ensure that all packets are brought into the processing core’s L1d cache. For the Ensō echo server, we use an RX/TX Ensō Pipe, which lets it echo packets without copies.

Figure 8 compares the packet rate for Ensō and DPDK for different numbers of cores. Even with a single core, Ensō is bottlenecked by Ethernet, achieving 148.8 Mpps. In contrast, the E810 with DPDK achieves 59 Mpps with a single core and does not scale beyond two cores, where it peaks at 88 Mpps. Beyond two cores, the experiments with the E810 are bottlenecked by PCIe bandwidth, which is insufficient for transferring packet data and descriptor metadata (§7.2.4). As a result, the number of packets dropped by the E810 NIC increases as we increase the number of cores, and the zero-loss throughput *decreases* beyond two cores.

### 7.2.2 Packet Forwarding with Copies

As we discussed in §3, Ensō’s streaming interface targets applications that process received data in order. With Ensō, applications that multiplex and demultiplex data, such as virtual switches [32, 67], need to copy packets to forward them to their destination. However, as we will see next, Ensō outperforms the E810 even in this scenario.

To quantify the overhead of multiplexing and demultiplexing data, we implement a simple packet forwarding application that swaps MAC addresses and copies incoming packets to a different TX Ensō Pipe. We compare this application against an equivalent zero-copy DPDK implementation. Figure 9 shows the throughput when forwarding packets of different sizes using a single CPU core. Packet copies add overhead to Ensō, which can no longer forward 64-byte packets at 100 Gbps (148.8 Mpps). However, Ensō’s throughput with 64-byte packets (91.7 Gbps) is still more than 2× that achieved by the E810 with DPDK *without* copies (39.6 Gbps). For other packet sizes, Ensō achieves line rate. We also evaluate the throughput for the same application when sending packets from a CAIDA trace [13]. For this trace Ensō’s throughput is 92.6 Gbps, compared to 72.6 Gbps for the E810 with DPDK without copies.

This result came as a surprise to us as our goal with Ensō was never to target multiplexing/demultiplexing in software.

	Ensō	Chaotic Ensō	E810
Throughput	100.0 Gbps	38.0 Gbps	41.1 Gbps
Rate	148.8 Mpps	56.5 Mpps	61.1 Mpps
L1d miss/total	22.8M/11,597M (0.2%)	557M/3,949M (14%)	1,357M/21,339M (6%)
L2 miss/total	2.1M/22.8M (9%)	382M/558M (68%)	747M/1,357M (55%)
LLC miss/total	541/2.1M (0.03%)	348k/382M (0.09%)	9,086/747M (0.001%)

**Table 1:** Effect of chaotic memory accesses on throughput and cache misses. Results are the average of 10 runs, each lasting 10 s.

It also puts into question the usefulness of a packetized interface, as its overheads can be greater than those imposed by packet copies.

Next, we use more detailed microbenchmarks that help explain where Ensō’s performance improvement comes from.

### 7.2.3 Effect of Chaotic Memory Accesses on Cache

We now show that placing messages sequentially in an Ensō Pipe is important for Ensō’s performance. As we discussed previously, not using sequential buffers results in chaotic memory accesses, which reduces the effectiveness of hardware prefetchers (e.g., streaming prefetcher [36]). To evaluate this claim, we built a modified version of Ensō, which we call Chaotic Ensō, that changes the gaps in memory between subsequent messages. We compute the gap deterministically based on the message’s current position in an Ensō Pipe, ensuring that we add no additional software overhead when using Chaotic Ensō.

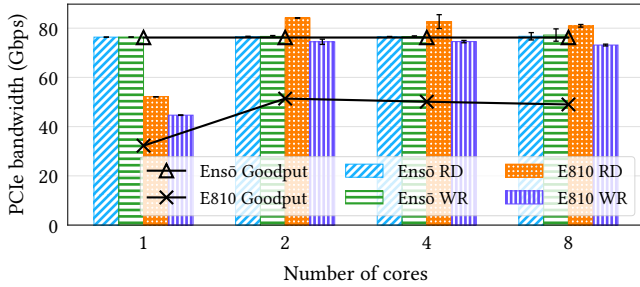
We benchmarked Ensō, Chaotic Ensō, and the E810 NIC using a program that receives and increments packets (but does not send them back). We measured zero-loss throughput, and cache miss rates at this throughput.

We show the results in Table 1. Observe that Chaotic Ensō achieves a lower throughput of 56.5 Mpps than even the E810. The number of cache misses reveals why: despite processing fewer packets per second (and hence having fewer cache accesses), Chaotic Ensō has an order-of-magnitude (557 million vs. 22.7 million) more L1d cache misses than Ensō. This, in turn, leads to an order of magnitude (558 million vs. 22.8 million) more accesses to L2 cache, thus increasing packet processing overheads. We also observe that the E810 has more cache accesses than either Ensō or chaotic Ensō, this is because it uses a descriptor per packet and thus requires the application to read more data. Finally, we observed that LLC misses were rare in all three configurations.

### 7.2.4 PCIe Bandwidth

Next, we evaluate the importance of reducing packet metadata by eliminating descriptors. We do so by measuring PCIe bandwidth when using the echo server described in §7.2.1 with 64-byte packets. In what follows, PCIe writes refer to DMA transfers from NIC to host memory, while PCIe reads refer to DMA transfers from the host memory to the NIC.

Unlike other microbenchmarks, we do not limit ourselves to zero-loss throughput for this experiment, and instead send



**Figure 10:** PCIe bandwidth utilization for read (RD) and write (WR) transactions. The bars represent the overall PCIe bandwidth utilization, while the lines represent the goodput, i.e., the amount of PCIe bandwidth used to transmit packet data. Ensō uses little PCIe bandwidth for metadata, causing it to achieve a higher goodput while consuming less overall PCIe bandwidth.

packets at line rate. This ensures that software overheads do not limit observed PCIe bandwidth, since drops due to queuing do not reduce throughput. We measure PCIe bandwidth and the rate at which the packet generator receives packets. We use this to calculate the fraction of PCIe bandwidth used for actual packet data (goodput).<sup>5</sup>

We report the results in Figure 10, where we show both PCIe goodput and total PCIe bandwidth for read (RD) and write (WR) directions. We draw four conclusions from it:

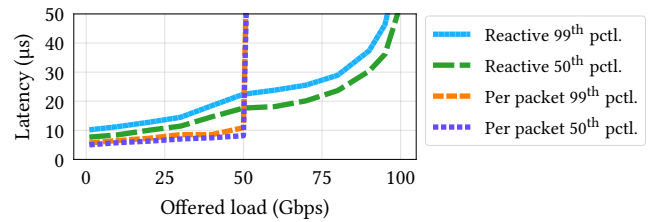
1. With one core, the E810 is CPU-bound, and the NIC drops incoming packets because of a lack of buffers in host memory. This reduces goodput and PCIe bandwidth utilization.
2. Beyond two cores, the E810 becomes PCIe bound, and consumes up to 84.1 Gbps of PCIe read bandwidth. This is close to 85 Gbps, the theoretical limit for PCIe Gen3 x16 with 64-byte transfers [62].
3. Even though the E810 has a lower goodput, it consumes more PCIe bandwidth due to metadata. This overhead accounts for up to 39% of the PCIe read bandwidth. In contrast Ensō’s metadata overhead remains below 1.2%.
4. Although barely noticeable in the plot, Ensō’s PCIe write bandwidth utilization increases as we increase the number of cores: going up from 76.4 Gbps (one core) to 77.2 Gbps (eight cores). This is because the NIC sends reactive notifications more frequently when software consumes packets faster.

While newer PCIe generations, including PCIe Gen 4, have higher capacity and will no longer be a bottleneck for 100 Gbps traffic, they will continue to be a problem for NICs that have multiple 100 Gbps interfaces and for future 400 and 500 Gbps NICs. Even with PCIe Gen 5 and a 400 Gbps NIC, the ratio of PCIe to ethernet bandwidth remains the same as in our setting with PCIe Gen 3.

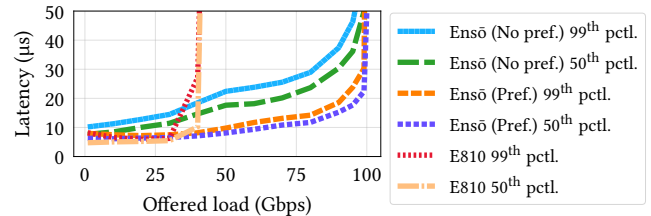
### 7.2.5 Reactive Notifications and Latency

As we discussed in §4.2, Ensō is able to reduce metadata overhead by sending notifications reactively. We measure

<sup>5</sup>The maximum goodput achievable with 64-byte packets and 100 Gb Ethernet is 76.19 Gbps, since Ethernet adds 20 bytes of overhead per packet.



**Figure 11:** RTT for different loads when using a notification per packet or reactive notifications without notification prefetching.



**Figure 12:** RTT for different loads for the E810 as well as Ensō with and without notification prefetching.

the impact reactive notifications have on throughput and latency, by comparing Ensō’s performance (reactive) to that of a variant of Ensō (per-packet) that sends a notification for each packet. We again reuse the echo server from previous microbenchmarks for this.

Figure 11 shows the RTT (50<sup>th</sup> and 99<sup>th</sup> percentiles) as we increase load for both cases. While reactive notifications can sustain up to 100 Gbps of offered load, a design using per-packet notifications can only sustain 50 Gbps. However, reactive notifications also add latency with increased load.

We use notification prefetching to minimize latency under high loads.<sup>6</sup> When using notification prefetching, the software explicitly sends the NIC a request for notifications pertaining to the next Ensō Pipe, while consuming data from the current Ensō Pipe. This effectively doubles the number of notifications that the NIC sends to software at a high rate but ensures that the software does not need to wait for a PCIe RTT before processing the next Ensō Pipe.

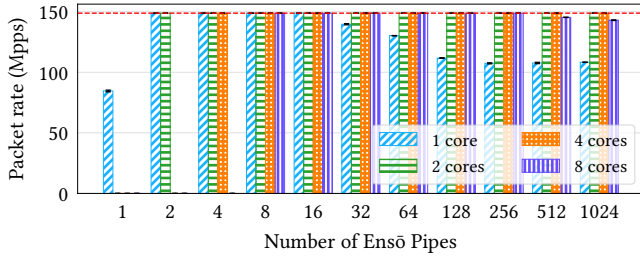
Figure 12 shows the RTT with an increasing load for Ensō with and without notification prefetching and for an E810 NIC with DPDK. We observe that notification prefetching significantly reduces Ensō’s latency, and allows us to achieve latency comparable to the E810, while still sustaining 100 Gbps.

### 7.2.6 Sensitivity Analysis

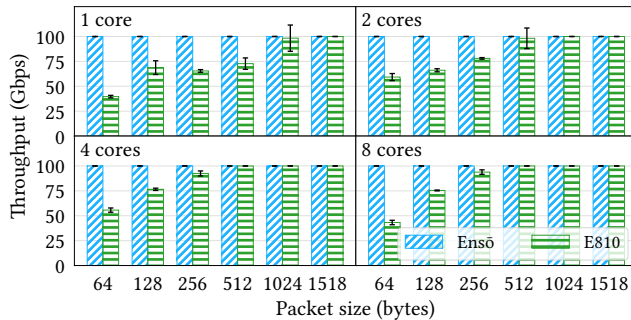
Finally, we use microbenchmarks to evaluate Ensō’s sensitivity to different configuration parameters:

**Impact of the number of Ensō Pipes:** We measure the impact of increasing the number of Ensō Pipes by varying the number of active Ensō Pipes, and using a workload where each incoming packet goes to a different Ensō Pipe. We partition Ensō Pipes evenly across all cores. Our results in Figure 13 show that (a) we need at least two Ensō Pipes to

<sup>6</sup>By default Ensō does not prefetch notifications. Latency-sensitive applications may enable notification prefetching at compile time.



**Figure 13:** Packet rate for different numbers of Ensō Pipes and core counts. The dashed line represents the 100 Gb Ethernet limit.



**Figure 14:** Sensitivity analysis with different numbers of cores and packet sizes. Ensō is bottlenecked by 100 Gb Ethernet in all scenarios.

achieve line rate, since this allows us to mask notification latency; and (b) that throughput drops when a core has more than 32 Ensō Pipes, or eight cores have more than 512. Note that this is a pessimal workload, and realistic workloads are likely to perform better even with many queues [30].

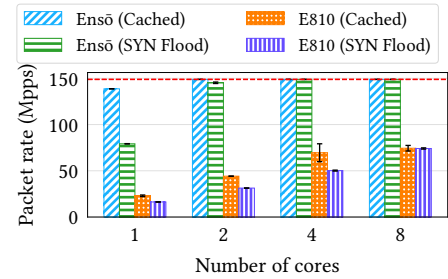
**Impact of packet sizes and cores:** In Figure 14 we measure the impact of varying packet size and number of cores, and find that Ensō can always sustain full line rate, regardless of packet size and core count.

### 7.3 Application Benchmarks

We now evaluate how Ensō impacts the performance of real applications. We ported four different applications to use both DPDK and Ensō. These applications represent three classes of network-intensive applications (raw packets, message-based, and streaming) that we expect to be used with Ensō: Google’s Maglev Load Balancer [23], a network-telemetry application based on NitroSketch [54], MICA Key Value Store [52], and a log monitor inspired by AWS Cloud-Watch Logs [6]. To enable a fair comparison, we use the same processing logic for both DPDK and Ensō-based implementations, changing only the wrapper code used to send and receive packets. Moreover, we only enable simple traditional offloads on the NIC, e.g., RSS, Flow Director, and checksum, for both Ensō and DPDK. We expect Ensō to perform even better with more offloads on the NIC (§6).

#### 7.3.1 Maglev Load Balancer

We implemented Google’s Maglev load balancer [23] as follows. We replicate the consistent-hashing algorithm pro-



**Figure 15:** Packet rate for the Maglev load balancer under two types of workloads. The dashed line is the 100 Gb Ethernet limit.

posed in the Maglev paper, caching recent flows in a flow table, as also suggested in the paper. The load balancer ultimately determines a backend server for every incoming packet, rewriting the packet’s destination IP to that of the chosen backend server. To steer packets among different cores, we use a hash of the 5-tuple (RSS) in both systems. We evaluate our implementation using two extreme types of workloads: one with only 16 flows, which means that packets always hit the flow cache; and another with a SYN flood, which means that packets always miss the flow cache. In both cases, we use small 64-byte packets as Maglev is motivated by the need to load balance small requests [23, §3.2]. For Ensō, we use unified RX/TX Ensō Pipes to avoid copying the packet when forwarding it back.

Figure 15 shows the packet rate with both the E810 NIC using DPDK and Ensō as we scale the number of cores. With a single core and the cached workload, Ensō achieves a packet rate of 138 Mpps, approximately 6× the 23 Mpps achieved by the E810. With the SYN flood workload, Ensō achieves 79 Mpps, approximately 5× the 16 Mpps achieved by the E810.<sup>7</sup> With four cores, Ensō becomes bottlenecked by Ethernet for both workloads; and with eight cores, the DPDK implementation becomes bottlenecked by PCIe (§7.2.4).

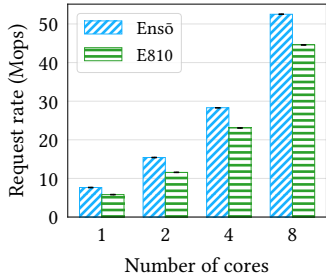
#### 7.3.2 Network Telemetry

Sketching algorithms are popular primitives for many network telemetry tasks (e.g., heavy-hitter detection, flow count estimation) because of their small memory footprint and theoretical accuracy guarantees. NitroSketch [54] is a sketching framework that enables *software sketches* to achieve high performance on commodity servers without sacrificing accuracy. To evaluate this class of applications, we implemented a Count-Min Sketch (CMS) using the NitroSketch framework and Ensō. As in Maglev, we use unified RX/TX Ensō Pipes.

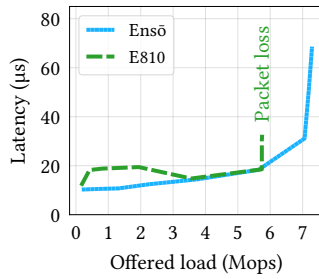
We benchmarked our implementation using two workloads: 64B packets (emulating the stress-test performed in [54]), and a busy period sampled from the 2016 CAIDA Equinix 10G dataset [13], with an average packet size of 462B.

<sup>7</sup>We note that DPDK’s packet rate of 16 Mpps with a single core is in fact a good packet rate for DPDK. For instance, NetBricks’ Maglev implementation achieves 9.2 Mpps with a single core [64]. We attribute the improvement in our DPDK numbers to NetBricks’ unoptimized implementation and our use of a newer CPU with better single-thread performance.

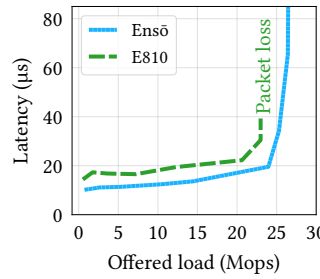




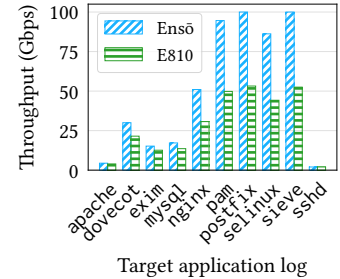
**Figure 16:** MICA throughput with 8B keys and values.



**Figure 17:** Mean RTT for MICA as a function of offered load (1 core).



**Figure 18:** Mean RTT for MICA as a function of offered load (4 cores).



**Figure 19:** Log monitor throughput for different target applications.

With a single core, Ensō sustains a zero-loss throughput of 81.5 Gbps (121 Mpps), approximately 3.5× that achieved by the E810 (22.8 Gbps or 33.9 Mpps) on 64B packets. On the CAIDA trace, Ensō achieves 94.9 Gbps, a 21% improvement over the E810 (78.3 Gbps); the application remains compute-bound in this setting, but Ensō shrinks the fraction of time spent performing network I/O, which improves performance.

### 7.3.3 MICA Key-Value Store

MICA [52] is a state-of-the-art key-value (KV) store which is also a popular benchmark in the literature [34,46,47,51,69,79]. Different from Maglev and the Network Telemetry application, that operate on raw packets and forward them back with modifications, MICA represents a typical message-based application whose responses must be constructed separately from the incoming request. Also different from these applications, latency is typically more critical for key-value stores [7]. MICA also entails significantly more work per packet (in terms of both compute and memory accesses) and is, therefore, less likely to become network-bound.

For the following experiments, we set the size for both keys and values to 8B (corresponding to the ‘tiny’ configuration in [52]). We report results for operations skewed 50% towards GET requests and with a uniform distribution of key popularity, but these generalize to other configurations as well. We use the same throughput metric as described in [52] (tolerating <1% loss at the NIC), and the same methodology for measuring end-to-end latency (the client tags each request with an 8B timestamp, then computes latency based on the arrival time of the corresponding response).

Figure 16 shows the steady-state throughput in millions of operations per second (Mops) achieved by MICA for both E810 with DPDK and Ensō for different numbers of cores. With a single core, Ensō achieves 7.65 Mops, a 31% improvement over the E810.<sup>8</sup> While this might seem modest at first (compared to the 6× speedup on Maglev), note that MICA is significantly more compute- and memory-intensive than Maglev. Thus, while DPDK adds considerable CPU overhead, it corresponds to a smaller fraction of the overall compute time.

<sup>8</sup>For consistency, all the MICA experiments use the original, unmodified MICA client implemented with DPDK. We observe even better performance when using a MICA client ported to Ensō (up to 47% improvement in throughput).

With 2 and 4 cores, we see throughput speedups of 33% and 23%, respectively. At 8 cores, the bottleneck moves to a different part of the system (i.e., memory). We report numbers for the ‘tiny’ configuration since it represents a significant fraction of requests found in real workloads [4, 57], while also being the most challenging workload for MICA. We also tested other configurations with larger keys and values, obtaining up to 29% improvement for the ‘small’ workload (16B keys and 64B values) and up to 12% for the ‘large’ workload (128B keys and 1024B values).

We also evaluate latency, plotting the average request latency as a function of the offered load when using a server with a single core (Figure 17) or four cores (Figure 18). For both configurations, we find that Ensō outperforms the E810 in terms of both throughput and latency. With a single core, Ensō reduces latency by up to 8 μs (43% reduction) before the queues start to build up and, with four cores, Ensō reduces latency by up to 6 μs (36% reduction).

### 7.3.4 Log Monitor

To understand Ensō’s effect on streaming applications, we implemented a log monitor. The log monitor is inspired by AWS CloudWatch Logs [6], which lets users centralize logs from different hosts and craft queries to look for specific patterns and raise alarms. Like, AWS CloudWatch Logs Insights [5], the log monitor also supports Hyperscan [84] to search for multiple regular expressions. We use Hyperscan in streaming mode for both Ensō and DPDK implementations. To feed the system, we use MTU-sized packets, carrying system logs extracted from long-running Linux hosts. We also configured the log monitor to look for regular expressions extracted from Fail2ban [45]. We run experiments targeting each of the ten most popular applications supported by Fail2ban according to the Debian package statistics [16].

Figure 19 shows the throughput we achieve when targeting each of the ten applications. Performance is dictated primarily by the number and complexity of the regular expressions that are required by each target. Ensō’s throughput is higher across all targets but the gap is more noticeable for those with simpler or fewer regular expressions, with almost double the throughput when targeting postfix, selinux, or sieve. The reasons for Ensō’s improvement in performance are twofold: First, Hyperscan performs better when



larger chunks of data are handed to it at once. With Ensō, we can invoke Hyperscan with large chunks of contiguous logs delivered from the NIC but with DPDK we need to invoke Hyperscan for every DPDK mbuf. Second, as demonstrated in §7.2.3, Ensō’s memory access patterns are sequential, making better use of the CPU prefetcher.

## 8 Related Work

**Direct application access:** While giving applications direct access to the NIC has been a common theme of research for more than three decades [8, 22, 26, 33, 50, 66, 73, 75, 80, 87, 88], most work accepts the NIC interface as a given and instead look at how to optimize the software interface exposed to applications. A notable exception is Application Device Channels [22], which gives control of the NIC to the kernel while giving applications independent access to different queues. We take inspiration from it in the way that we allow multiple applications to share the same NIC.

**Alternative NIC interfaces:** There are also proposals that try to address some of the performance and abstraction issues that we highlighted for the packetized interface.

In terms of performance, Nvidia MLX 5 NICs [20] provide a feature named Multi-Packet Receive Queue (MPRQ) that can potentially reduce PCIe RD bandwidth utilization with metadata by allowing software to post multiple packet buffers at once. However, this is not enough to completely avoid PCIe bottlenecks as the NIC still needs to notify the arrival of every packet, consuming PCIe WR bandwidth. Another proposed change to the NIC interface is Batched RxList [70]. This design aggregates multiple packets in the same buffer as a way to allow descriptor ring buffers to be shared more efficiently by multiple threads, which in turn could help them avoid the leaky DMA problem [81].

In terms of abstraction, U-Net [83] and, more recently, NICA [27] allow the NIC to exchange application-level messages directly. U-Net proposes a communication abstraction that resembles part of what is now libibverbs (RDMA) [53] and NICA uses a similar mechanism named “custom rings.” However, similar to the packetized interface, both U-Net and NICA use descriptors and scattered buffers and, as such, inherit its performance limitations.

**Application-specific hardware optimizations:** Prior work has optimized the NIC for specific applications. FlexNIC [49] quantifies the benefits that custom NIC interfaces could have to different applications. NIQ [29] implements a mechanism to reduce latency for minimum-sized packets by using MMIO writes to transmit these packets. It also favors MMIO reads over DMA writes for notifying incoming packets. NIQ’s reliance on MMIO means that it is mostly useful for applications that are willing to vastly sacrifice throughput and CPU cycles to improve latency. nm-

NFV [69] stores packet payloads on NIC memory, sending only the packet headers inlined inside descriptors, which is useful for network functions that only need to modify the header. This is orthogonal to Ensō’s interface changes and could also be used in conjunction with it.

**Application-specific software optimizations:** Some proposals avoid part of the overheads of existing NICs with application-specific optimizations in software. TinyNF [68] is a userspace driver optimized for network functions (NFs). It relies on the fact that NFs typically retransmit the same packet after processing. It keeps the set of buffers in the RX and TX descriptor rings fixed, reducing buffer management overheads. eRPC [46] is an RPC framework that employs many RPC-specific optimizations. For instance, it reduces transmission overheads by ignoring completion notifications from the NIC, instead relying on RPC responses as a proxy for completions. FaRM [21] is a distributed memory implementation. It uses one-sided RDMA to implement a message ring buffer data structure that has some similarities to an Ensō Pipe. However, different from an Ensō Pipe, FaRM’s message buffer is not opaque (enforcing a specific message scheme), must be exclusive to every sender, and lacks a separate notification queue (requiring the receiver to fill the buffer with zeros and to probe every buffer for new messages).

## 9 Conclusion

Ensō provides a new streaming abstraction for communication between software and NICs. It is better suited to modern NICs with offloads and improves throughput by up to 6× by being more cache- and prefetch-friendly and by reducing the amount of metadata transferred over the IO bus. While this paper focused on using Ensō for NIC-to-software communication, we believe that a similar approach might also apply to other I/O devices and accelerators, and we hope to explore this in future work.

## Acknowledgments

We thank our shepherd, Jon Crowcroft, and the anonymous OSDI ’23 reviewers for their great comments. We thank Francisco Pereira and Adithya Abraham Philip for their comments on this and earlier drafts of this paper, and Ilias Marinos for the discussions and feedback regarding applications. We also thank the people from Intel and VMware that gave us feedback throughout this work, including Roger Chien, David Ott, Ben Pfaff, Yipeng Wang, and Gerd Zellweger.

This work was supported in part by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, by a Google Faculty Research Award, and by ERDF through the COMPETE 2020 program as part of the project AIDA (POCI-01-0247-FEDER-045907). Nirav Atre was supported by a CyLab Presidential Fellowship.

## References

- [1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks, HotNets '22*, pages 198–204, New York, NY, USA, 2022. 2.2
- [2] Amazon. DPKDK driver for elastic network adapter (ENA). <https://github.com/amzn/amzn-drivers/tree/master/userspace/dpdk>, 2022. 2
- [3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlauff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI '20*, pages 93–109, Santa Clara, CA, February 2020. 1
- [4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12*, pages 53–64, New York, NY, USA, 2012. 1, 2.2, 7.3.3
- [5] AWS. CloudWatch Logs Insights query syntax, 2022. [https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL\\_QuerySyntax.html](https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html). 7.3.4
- [6] AWS. What is Amazon CloudWatch Logs?, 2022. <https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html>. 1, 7.3, 7.3.4
- [7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017. 4.2, 7.3.3
- [8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 49–65, Broomfield, CO, October 2014. 8
- [9] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosf, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20*, pages 753–768, November 2020. 1
- [10] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4T: A fast and flexible FPGA-based full-stack TCP acceleration framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, Orlando, FL, USA, 2023. 6
- [11] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, March 1999. 7.1
- [12] Broadcom. BNXT poll mode driver. <https://doc.dpdk.org/guides/nics/bnxt.html>, 2022. 2
- [13] CAIDA. Anonymized internet traces 2016. [https://catalog.caida.org/dataset/passive\\_2016\\_pcap](https://catalog.caida.org/dataset/passive_2016_pcap). 3, 7.2.2, 7.3.2
- [14] Chelsio Communications. Terminator 5 ASIC, 2021. <https://www.chelsio.com/terminator-5-asic/>. 1, 6
- [15] Jonathan Corbet. Ringing in a new asynchronous I/O API, 2019. <https://lwn.net/Articles/776703/>. 2
- [16] Debian. Debian popularity contest: Statistics by source packages (sum) sorted by fields, 2022. [https://popcon.debian.org/source/by\\_inst](https://popcon.debian.org/source/by_inst). 7.3.4
- [17] Roman Dementiev et al. Processor Counter Monitor (PCM), 2022. <https://github.com/opcm/pcm>. 7.1
- [18] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, pages 15–28, New York, NY, USA, 2009. 2.2
- [19] DPKDK. Data Plane Development Kit, 2022. <https://dpdk.org>. 1, 2
- [20] DPKDK. NVIDIA MLX5 ethernet driver, 2022. <https://doc.dpdk.org/guides/nics/mlx5.html>. 1, 8
- [21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation, NSDI '14*, pages 401–414, Seattle, WA, April 2014. 8

- [22] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 2–13, New York, NY, USA, 1994. 1, 4.1.2, 8
- [23] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 523–535, Santa Clara, CA, March 2016. 1, 7.3, 7.3.1
- [24] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, 2015. 7.1, B
- [25] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '19, pages 1–12, Cambridge, UK, 2019. IEEE. 4.1, A
- [26] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. 8
- [27] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference*, ATC '19, pages 345–362, Renton, WA, July 2019. 1, 8
- [28] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohita, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, April 2018. 2.1
- [29] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *2013 USENIX Annual Technical Conference*, ATC '13, pages 333–346, San Jose, CA, June 2013. 8
- [30] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Gironi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet order matters! Improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 807–827, Renton, WA, April 2022. 7.2.6
- [31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. 1
- [32] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. 2.1, 3, 6, 7.2.2
- [33] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. 8
- [34] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 239–256, July 2021. 7.3.3
- [35] Intel. Intel data direct I/O technology (Intel DDIO): A primer. Technical report, Intel, February 2012. 2.2
- [36] Intel. Intel 64 and IA-32 architectures optimization reference manual. Technical Report 248966-045, Intel, 2022. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. 7.2.3
- [37] Intel. Intel Core i7-7820X X-series Processor, 2022. <https://ark.intel.com/content/www/us/en/ark/products/123767/intel-core-i77820x-xseries-processor-11m-cache-up-to-4-30-ghz.html>. 7.1



- [38] Intel. Intel Core i9-9960X X-series Processor, 2022. <https://ark.intel.com/content/www/us/en/ark/products/189123/intel-core-i99960x-xseries-processor-22m-cache-up-to-4-50-ghz.html>. 7.1
- [39] Intel. Intel Ethernet Controller E810. Technical Report 613875-006, Intel, March 2022. 1, 2, 2.1, 4.1, A, B
- [40] Intel. Intel Ethernet Network Adapter E810-CQDA2, 2022. <https://ark.intel.com/content/www/us/en/ark/products/210969/intel-ethernet-network-adapter-e8102cqda2.html>. 7.1
- [41] Intel. Intel infrastructure processing unit (Intel IPU) platform (codename: Oak Springs Canyon), 2022. <https://www.intel.com/content/www/us/en/products/platforms/details/oak-springs-canyon.html>. 6
- [42] Intel. Intel Stratix 10 MX 2100 FPGA, 2022. <https://ark.intel.com/content/www/us/en/ark/products/210297/intel-stratix-10-mx-2100-fpga.html>. 7.1, B
- [43] Intel. Top-down microarchitecture analysis method. <https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html>, 2022. 2.2
- [44] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 322–334, Virtual Event, 2020. IEEE Press. 1
- [45] Cyril Jaquier et al. Fail2ban, 2022. <https://www.fail2ban.org/>. 7.3.4
- [46] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 1–16, Boston, MA, February 2019. 7.3.3, 8
- [47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 185–201, Savannah, GA, November 2016. 7.3.3
- [48] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. 1
- [49] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. 6, 8
- [50] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. 8
- [51] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-Memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. 7.3.3
- [52] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 429–444, Seattle, WA, April 2014. 1, 2.2, 7.1, 7.3, 7.3.3
- [53] Linux RDMA. RDMA core userspace libraries and daemons, 2022. <https://github.com/linux-rdma/rdma-core>. 8
- [54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. NitroSketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 334–350, New York, NY, USA, 2019. 1, 7.3, 7.3.2
- [55] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 270–282, New York, NY, USA, 2020. 2.2



- [56] Marvell. DPKD marvell. <https://github.com/MarvellEmbeddedProcessors/dpdk-marvell>, 2022. 2
- [57] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, pages 243–262, New York, NY, USA, 2021. 1, 7.3.3
- [58] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jijaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference, SIGCOMM '22*, pages 753–766, New York, NY, USA, 2022. 1
- [59] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. HyperPlane: A scalable low-latency notification accelerator for software data planes. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '20*, pages 852–867, 2020. 4.1.2
- [60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18*, pages 221–235, New York, NY, USA, 2018. 6
- [61] Al Morton. RFC Errata, Erratum ID 412, RFC 2544, November 2006. <https://www.rfc-editor.org/errata/eid422>. 7.1
- [62] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18*, pages 327–341, New York, NY, USA, 2018. 2.2, 2
- [63] Nvidia. NVIDIA BlueField-3 DPU, 2022. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf>. B
- [64] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16*, pages 203–216, Savannah, GA, November 2016. 7
- [65] Perf. perf: Linux profiling with performance counters, 2022. <https://perf.wiki.kernel.org>. 7.1
- [66] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 1–16, Broomfield, CO, October 2014. 8
- [67] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI '15*, pages 117–130, Oakland, CA, May 2015. 2.1, 3, 6, 7.2.2
- [68] Solal Pirelli and George Candea. A simpler and faster NIC driver model for network functions. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20*, pages 225–241, November 2020. 2, 6, 8
- [69] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafir. The benefits of general-purpose on-NIC memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, New York, NY, USA, 2022. 7.3.3, 8
- [70] Boris Pismenny, Adam Morrison, and Dan Tsafir. ShRing: Networking with shared receive rings. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI '23*, Boston, MA, July 2023. 8
- [71] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, pages 1203–1216, New York, NY, USA, 2020. 1
- [72] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21*, pages 199–205, New York, NY, USA, 2021. 1
- [73] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference, ATC '12*, pages 101–112, Boston, MA, 2012. 2, 8

- [74] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A case for spraying packets in software middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 127–133, New York, NY, USA, 2018. [B](#)
- [75] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. [8](#)
- [76] Alireza Sanaee, Farbod Shahinfar, Gianni Antichi, and Brent E. Stephens. Backdraft: A lossless virtual switch that prevents the slow receiver problem. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 1375–1392, Renton, WA, April 2022. [4.1.2](#)
- [77] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with fine-grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 87–102, Renton, WA, April 2022. [1, 6](#)
- [78] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 708–721, New York, NY, USA, 2020. [6](#)
- [79] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The NeBuLa RPC-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 199–212, Virtual Event, 2020. IEEE Press. [7.3.3](#)
- [80] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. In *Conference Proceedings on Communications Architectures, Protocols and Applications*, SIGCOMM '93, pages 64–73, New York, NY, USA, 1993. [8](#)
- [81] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 283–297, Renton, WA, April 2018. [2.2, 8](#)
- [82] Amy Viviano. Introduction to receive side scaling, 2022. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>. [2.1, 6](#)
- [83] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. [8](#)
- [84] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 631–648, Boston, MA, February 2019. [7.3.4](#)
- [85] Keith Wiles et al. The Pktgen application, 2022. <https://pktgen-dpdk.readthedocs.io/>. [7.1](#)
- [86] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 206–212, New York, NY, USA, 2021. [1](#)
- [87] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference*, ATC '16, pages 43–56, Denver, CO, June 2016. [8](#)
- [88] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 195–211, New York, NY, USA, 2021. [1, 8](#)
- [89] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 483–495, New York, NY, USA, 2020. [1](#)

## Appendix A Rejected Designs for Pointer Updates

Unfortunately, the simplest alternatives for communicating pointer updates from the NIC to software perform poorly. We considered three potential alternative designs to let the NIC inform pointer updates to software. While these designs might seem to suit our needs at first, we ultimately discarded them since they perform poorly due to architectural details of PCIe or the CPU:

**MMIO synchronization:** The simplest design would be for the NIC to update the pointer values to its internal memory and make software periodically issue an MMIO read to fetch the latest value. Unfortunately, software-issued MMIO reads cannot be served from the cache, causing the CPU core to stall until the read request is sent and the response is received (two PCIe transfers). Additionally, PCIe serializes MMIO reads, further reducing performance.

**Shared memory synchronization:** A second simple alternative would be to dedicate an address in host memory to hold the pointer value. The NIC can issue a DMA write whenever it needs to update the pointer value. Software can then periodically poll the same address to figure out if the NIC advanced the pointer. The issue with this design is that it makes the software and the NIC contend for the same cache line, which requires a slow ownership transfer whenever the NIC or the CPU access the cache line. To verify this, we implemented this design and obtained a throughput of less than 5 Gbps when enqueueing 64-byte packets using the same setup as described in §7.

**Inline synchronization:** The last discarded design is motivated by traditional NIC descriptor ring buffers which contain constant-sized descriptors with metadata in a format defined by the NIC. These designs designate a bit of the descriptor as a “flag bit” [25, 39]. Initially, all the descriptor slots in the buffer have their flag bit zeroed. Whenever the NIC DMA writes a new descriptor to host memory, it overwrites the old ‘0’ flag with a ‘1’ flag. To figure out if a new descriptor is ready to be consumed, software simply needs to check if the next slot’s flag is set. After consuming a descriptor, software sets the flag back to ‘0.’ Because many descriptors are likely to be present in the buffer, this reduces the chance that software and the NIC will contend for the same cache line.

Since Ensō Pipes are opaque, implementing the same strategy used in the descriptor ring buffer is impossible. Even if we zeroed the entire buffer after the data is consumed, we do not know what data to expect in the buffer—the next incoming data might also be zero. Therefore we tested an alternative design: we picked a 128-bit random cookie, to make the chance of collision with incoming data negligible and placed it at the beginning of every cache line of the Ensō Pipe. Software now only needs to check if the next 128 bits

match the cookie. Unfortunately, filling the buffer with cookies whenever the data is consumed imposes considerable overhead. For this reason, this design worked well for small data transfers but poorly when using large chunks of data. This design also prevents software from detecting unaligned writes.

## Appendix B EnsōGen Packet Generator

EnsōGen is a software packet generator built on top of the Ensō NIC interface that achieves 100 Gbps with a single core and arbitrary packet sizes. Here we briefly describe how it operates and how we ensure that it is correct.

**Operation:** At startup EnsōGen reads a user-supplied pcap file and allocates enough Ensō Pipes to be able to fit all its packets. At run time, EnsōGen simply needs to round-robin among the pre-allocated Ensō Pipes enqueueing a *single* notification in order to transmit the *entire* 2 MB buffer content. This makes it trivial for EnsōGen to saturate the link with very little CPU overhead. Since transmission is cheap, EnsōGen spends most of its CPU cycles receiving packets. It parses every incoming packet to track the number of bytes and packets received.

**Simple offloads:** We implemented hardware support for timestamping and rate limiting, which helps EnsōGen achieve cycle-accurate precision while saving CPU cycles. These features are also commonly offered in existing NICs [39, 63] and are leveraged by some software packet generators [24]. When hardware timestamping is enabled, EnsōGen keeps a histogram in host memory with the RTT of every received packet with 5 ns granularity (the same precision as the hardware timestamper, which operates at 200 MHz). To spread the load equally among the RX Ensō Pipes regardless of the workload, EnsōGen also configures the hardware to direct packets to pipes in a round-robin fashion [74].

**Correctness:** We verified EnsōGen’s performance and rate-limiting capabilities using another in-house packet generator fully implemented on an FPGA, as well as using software counters, to ensure that the rate limited throughput always matches the specification.

## Artifact Appendix

### Abstract

The paper artifact is composed of Ensō’s hardware and software implementations, the applications used in the evaluation, the EnsōGen packet generator, as well as the code to automatically run most of the experiments. We also include documentation describing how to set up the environment, compile the code, synthesize the hardware, and use Ensō for other purposes—including a detailed description of the software API.

### Scope

The artifact has two main goals: The first is to allow the main claims in the paper to be validated. The second is to allow others to build upon Ensō for their own projects.

We include code to automatically reproduce Figures 8, 11, 12, 13, 14, and 15. We also include the source code for all the applications that we evaluate in §7.3 and for the EnsōGen packet generator.

### Contents

The artifact is split between two git repositories.

#### Ensō Repository

This repository includes Ensō’s source code as well as documentation and example applications. It is structured as follows:

**hardware/:** Source code for the hardware component and scripts to automatically generate all the required IPs.

**software/:** Source code for the software component, which includes both the library and the kernel module. It also includes example applications and EnsōGen under `software/examples`.

**frontend/:** Frontend to programmatically load and configure the hardware from Python as well as a command line interface based on this frontend.

**docs/:** Documentation detailing how to set up the system, compile the software and the hardware, and how to use Ensō’s primitives (RX Pipes, TX Pipes, and RX/TX Pipes) from an application.

#### Ensō Evaluation Repository

This repository includes code to automatically run experiments to verify the main claims in the paper and the applica-

tions that we evaluate in §7.3. Here we briefly describe the main files and directories:

**experiment.py:** Script to automatically run the experiments to verify the main claims in the paper.

**paper\_plots.py:** Script to produce all the plots in the paper.

**setup.sh:** Script to automatically setup the experiment environment.

**maglev/:** Maglev Load Balancer used in §7.3.1.

**nitrosketch/:** Network telemetry application based on NitroSketch used in §7.3.2.

**mica2/:** MICA Key-Value store used in §7.3.3.

**log\_monitor/:** Log monitor application used in §7.3.4.

### Hosting

Both repositories are hosted on GitHub and archived using Zenodo with a permanent DOI. The documentation contained in the Ensō Repository is also automatically deployed using GitHub actions for easy access.

#### Ensō Repository

- Repository: <https://github.com/crossroadsfpga/enso>
- Commit: [093dca77836f8e10409af7f0ec3b28232fc25f44](https://github.com/crossroadsfpga/enso/commit/093dca77836f8e10409af7f0ec3b28232fc25f44)
- Zenodo Archive: <https://zenodo.org/record/7860872>
- DOI: <https://doi.org/10.5281/zenodo.7860872>

#### Ensō Evaluation Repository

- Repository: [https://github.com/crossroadsfpga/enso\\_eval](https://github.com/crossroadsfpga/enso_eval)
- Commit: [1a100cb38577930b9124fc6fefced3f0a6da7da4](https://github.com/crossroadsfpga/enso_eval/commit/1a100cb38577930b9124fc6fefced3f0a6da7da4)
- Zenodo Archive: <https://zenodo.org/record/7860936>
- DOI: <https://doi.org/10.5281/zenodo.7860936>

#### Ensō Documentation

- Link: <https://enso.cs.cmu.edu>

### Requirements

Running Ensō requires a host equipped with an Intel Stratix 10 MX FPGA [42] and an x86-64 CPU. The software component also assumes that the host is running Linux. §7 details the exact environment we used in our experiments.